The connection between C++ template metaprogramming and functional programming

Doctoral dissertation
2013

Ábel Sinkovics
abel@elte.hu

Thesis advisor: **Zoltán Porkoláb, PhD**
Eötvös Loránd University, Faculty of Informatics,
1117 Budapest, Pázmány Péter sétány 1/C

ELTE IK Doctoral School
Doctoral program: Az informatika alapjai és módszertana
Head of the doctoral school: Dr. András Benczúr, academicer
Head of the doctoral program: Dr. János Demetrovics, academicer

The Project is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TAMOP 4.2.1./B-09/1/KMR-2010-0003).

Contents

Acknowledgements

This work would not have been done without the continuous support and invaluable advices of my thesis advisor, Zoltán Porkoláb. I'd also like to express my gratitude towards my wife, Kati for her understanding and endless patience.

Ez a munka nem készülhetett volna el témavezetőm, Porkoláb Zoltán folyamatos támogatása és értékes tanácsai nélkül. Emellett szeretném kifejezni hálámat feleségem, Kati iránt is megértéséért és végtelen türelméért.

Chapter I

Introduction

This dissertation introduces advanced techniques for C++ template metaprogramming supporting the developers and maintainers of applications and libraries implemented in C++. It assumes, that the reader is already familiar with the C++ programming language.

In 1994 Erwin Unruh demonstrated [75] that it is possible to execute algorithms using a C++ compiler as a side-effect of template instantiations. Programs based on this technique are called *C++ template metaprograms* and they form a Turing-complete sub-language of C++ [79]. Most developers don't work on template metaprograms directly, but use libraries that are based on template metaprograms. Since template metaprograms are executed at compile-time, a number of extensions to the C++ language can be implemented using them without having to change the compiler itself.

- For C++ libraries supporting a specific domain (database access, regular expressions, etc.) it is useful if domain-specific errors (database field type mismatches when reading or writing code, invalid regular expressions, etc.) can be caught at compile-time, instead of leaving them in the code and break the program execution at runtime. C++ template metaprograms make it possible to implement such verifications [18].

- Domain Specific Languages (DSL) [16] are more and more popular. They are small languages targeting one specific domain. For that domain, they are better suited than other languages, but they are not useful in other domains. There are widely used and well known domain specific languages, such as SQL or the syntax of regular expressions. DSLs are used together with one (or more) general purpose language in a way, that the parts of the program describing the domain specific logic are implemented in the DSL, while the rest of the code is implemented in the general purpose language. When the code snippets

written in the DSL are embedded into a source code written in another (in most cases general purpose) language, they are called Embedded DSLs, or EDSLs. Template metaprograms can be used to embed such languages into C++ [65].

- Developers have to write repetitive code from time to time, where there are minor differences between the repetitions. In most cases, such code snippets are implemented by copying and updating another one. Template metaprograms provide solutions for making the C++ compiler generate such code snippets [1].

- C++ compilers optimise the code to make it run faster or consume less memory. However, the options for the compiler to optimise are limited by the fact that the optimised code has to work the same way as the original one. In many cases, further optimisations could be done based on extra knowledge about the domain of the application. The compiler is not aware of the domain and can not implement these optimisations, however, they can be implemented using template metaprogramming techniques [78, 80, 77].

- When new features are added to the C++ language or when someone would like to try a new idea for a language feature out, the compiler needs to be updated, which is not always easy or even possible. However, many such features can be simulated using template metaprogramming techniques [83].

C++ template metaprograms can be implemented following the standard, thus all of the standard compliant compilers can understand and execute them.

Templates in C++ were not designed to form a Turing-complete sublanguage of C++, but they are complex enough to make executing algorithms at compile-time possible. Using the template instantiation logic of the compiler to execute custom algorithms has drawbacks, since this capability has never been a design goal. When someone writes template metaprograms, he uses compile-time functions and data-structures, however, he implements them by template classes, `typedef`s and inheritance. This makes it difficult to write, read and maintain template metaprograms.

I.1 Motivation

The connection between the functional programming paradigm and C++ template metaprogramming is well known [37, 10, 40, 15, 52, 68, 19, 20]. There are a number of similarities between the logic of C++ template metaprogramming and functional languages, like Haskell. For C++ code executed at runtime there are libraries supporting functional programming [36] but in template metaprogramming current approaches [1, 25, 3] try to simulate imperative languages and libraries, such as the Standard Template Library of C++ [32, 41] and most of them does not take advantage of the functional paradigm.

I.2 Structure of the dissertation

This dissertation presents different approaches for providing a better syntax and abstractions for template metaprogramming than what is available now. All of the techniques discussed are based on the C++ standard, they can be used with any standard compliant compiler. None of them requires external tools. The logical structure of the dissertation is presented in figure I.1. The cloud represents the initial idea of the dissertation. The rounded rectangles represent the theses. The section numbers in which the topics are discussed are added to the diagram in the small rectangles. The rectangles with thick grey borders represent the benefits of following the topics discussed in this dissertation.

After an introduction to template metaprogramming and presenting the current practice in chapter II, chapter III presents how to provide a number of *language elements of Haskell* in C++ template metaprogramming and how to use them. Given how strongly it affects the way programs are implemented, this chapter begins with a discussion of the *evaluation strategies* in C++ template metaprograms. After that it presents abstractions that simplify template metaprograms, such as *currying*, *algebraic data types*, *typeclasses*, *let* and *lambda expressions* and *pattern matching*.

Chapter IV presents how to implement *Monads*, an abstraction commonly used in functional programming languages [29, 51, 55, 56, 28, 48, 56] in C++ template metaprogramming using the techniques presented in chapter III. It presents a number of *monad instances* and presents how to implement them. Haskell provides a syntactic sugar, the *do syntax* for writing monadic code, which strongly improves the readability of the code. This chapter presents a way for simulating this notation in template metaprogramming to get the same benefits. It is also discussed, how it provides *list comprehension* in tem-

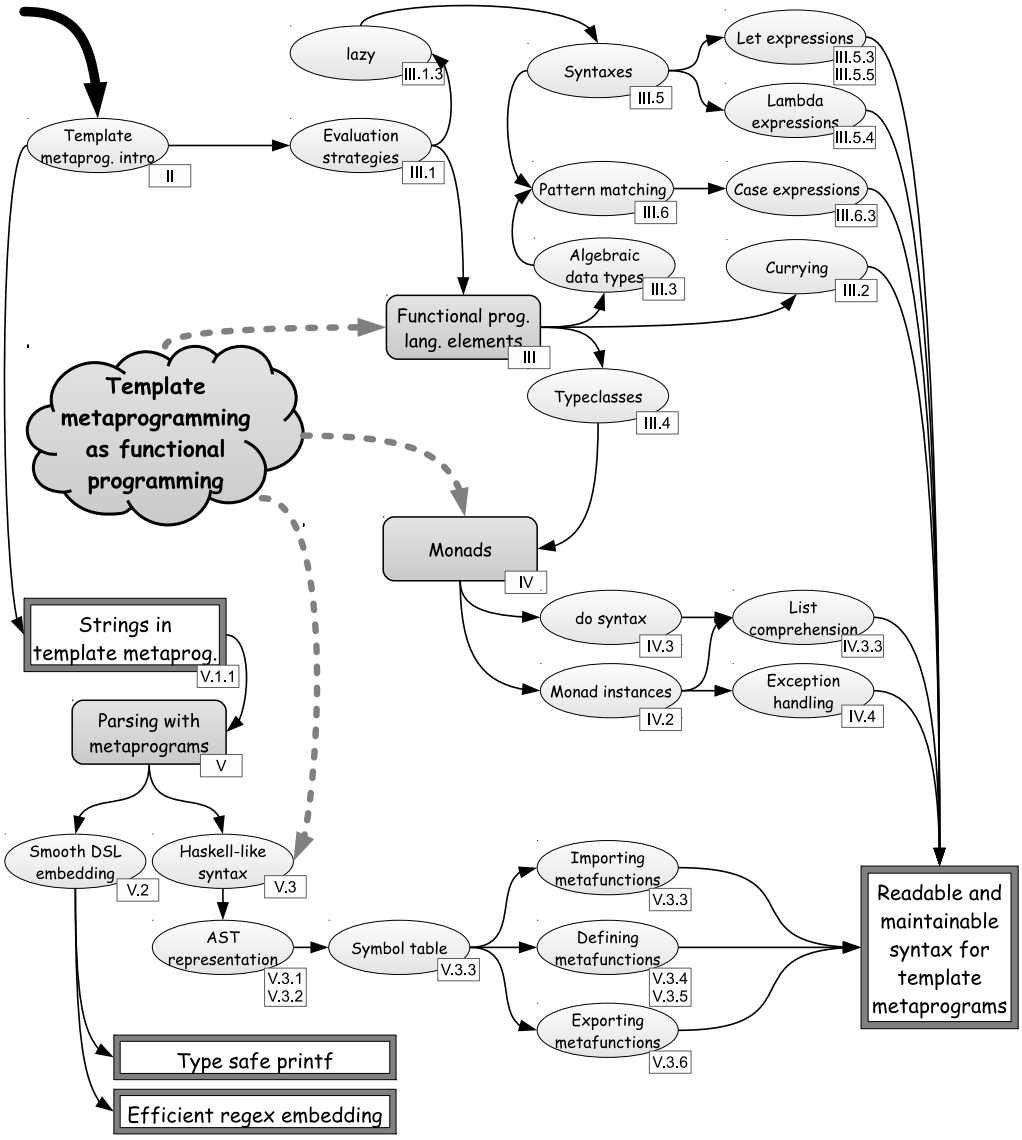

Figure I.1: Structure of the dissertation

plate metaprogramming. The chapter discusses, how to implement and use a generalisation of Haskell's *Either* type in C++ template metaprogramming to simulate *exception handling* and how to provide *try* and *catch blocks* for template metaprograms based on this.

These techniques support the authors of template metaprograms in following the functional programming paradigm and reasoning about template metaprograms as Haskell programs, however, as the resulting code has to be valid C++ code, there are syntactic limitations. Chapter V presents a technique for *parsing the content of string literals* by template metaprograms. It presents how to build a parser for a *Haskell-like language*, that parses function definitions written in that language and execute them as template metaprograms as part of the same compilation process. This makes it possible to write template metaprograms using a Haskell-like syntax, which is easier to read and maintain than the original syntax of template metaprogramming. Based on the technique for parsing embedded code snippets written in another language, it is possible to provide smooth *embedding of domain specific languages*. The chapter presents further examples for this usage as well.

All techniques presented in this dissertation are concluded in chapter VI. A reference implementation of the discussed techniques is also available as an open-source library [59].

I.3 Contributions

This section contains the list of contributions of this dissertation. Each contribution has a short description followed by a reference to a chapter or a section where it is discussed in detail.

Thesis 1: I have evaluated the connection between C++ template metaprogramming and functional programming languages. Based on the results I have developed methods for supporting template metaprogrammers using the functional paradigm explicitly. (chapter III)

Thesis 1.1: I have shown the importance of laziness in template metaprogramming and developed an automated adaption method to use non-lazy metafunctions in a lazy way. (section III.1)

Thesis 1.2: I developed a method for effective implementation of currying in C++ template metaprogramming. (section III.2)

Thesis 1.3: I have developed a method for representing Haskell-like algebraic data-types in C++ template metaprogramming. (section III.3)

Thesis 1.4: I have developed a method for representing Haskell typeclasses in C++ template metaprogramming. (section III.4)

Thesis 1.5: I have developed a method to handle template metaprogramming expressions as first class citizens, ie. they can be stored, passed as parameters or returned by functions. This method enables the implementation of let expressions and provides a more convenient way of implementing lambda expressions than what Boost.MPL's lambda expression implementation, a widely used solution offers. (section III.5)

Thesis 1.6: I have implemented an alternative method for pattern matching in C++ template metaprogramming. This enables the implementation of case expressions. (section III.6)

Thesis 2: I have developed a method for implementing monads and a Haskell-like do syntax in C++ template metaprogramming and evaluated how a number of different monad variations available in Haskell can be implemented using this method. Based on this I have developed a method for simulating exception handling in C++ template metaprograms. (chapter IV)

Thesis 2.1: I have developed a method for implementing monads in C++ template metaprogramming. (section IV.1)

Thesis 2.2: I have evaluated how a number of monads available in Haskell can be implemented using the approach presented in Thesis 2.1. (section IV.2)

Thesis 2.3: I have developed a method for implementing a Haskell-like do syntax in template metaprogramming. (section IV.3)

Thesis 2.4: I have developed a method for simulating exception handling in C++ template metaprogramming based on monads. (section IV.4)

Thesis 3: I have developed a method for implementing a parser generator library in C++ template metaprogramming. I have evaluated how it can be used for embedding domain specific languages into C++ and providing a more readable syntax for C++ template metaprogramming. None of these methods require external preprocessors. (chapter V)

Thesis 3.1: I have developed a method for turning string literals into character containers for C++ template metaprograms. Utilising this I have developed a method for implementing a parser generator library in C++. (section V.1)

Thesis 3.2: I have evaluated how parsers based on Thesis 3.1 can be used to embed domain specific languages into C++ without external preprocessors. (section V.2)

Thesis 3.3: I have developed a method based on Thesis 3.1 for providing a Haskell-like syntax for C++ template metaprograms without external preprocessors. (section V.3)

Chapter II

Introduction to template metaprogramming

This chapter introduces template metaprogramming by presenting a way of using template classes and template instantiations to calculate factorial numbers at compile-time. After that this chapter presents the basic concepts of template metaprogramming, such as *template metafunctions*, *boxed values*, *higher order metafunctions* and *selections*. This chapter discusses the connection between template metaprogramming and functional languages as well.

II.1 First example: factorial

The way template metaprogramming works can be demonstrated through a simple example: calculating the factorial of numbers at compile-time. This is not a real-world use case, however, it is good for demonstrating how to write programs running at compile-time and a number of problems one can face. The factorial of a number can be calculated at compile-time using a template class taking one template argument, the number to calculate the factorial of:

```
template <int N>
struct fact {
  static const int value = fact<N - 1>::value * N;
};
```

This template class has a `static int` constant member. The value of this constant member is the factorial of the template argument. It is calculated

using a recursive expression: the factorial of `N` is `N` times the factorial of `N - 1`. The factorial of `N - 1` is the value of the `value` member of the class `fact<N - 1>`. The instantiation of `fact<N>` triggers the instantiation of `fact<N - 1>` which triggers the instantiation of `fact<N - 2>` and it keeps recursing. This recursive chain of instantiations has to be stopped. The factorial of `0` is `1`. The chain of recursion can be stopped at `0` using *template specialisation* [76]. A specialisation has to be written for `fact<0>` that doesn't call `fact` recursively:

```
template <>
struct fact<0> {
  static const int value = 1;
};
```

Having this specialisation when the recursive instantiation chain tries to instantiate `fact<0>`, the compiler chooses the specialised version of the template class and doesn't recurse further.

The above example demonstrates how to use C++ templates to do calculations at compile-time. To be able to build more complicated programs that are executed at compile-time a convention is needed on how these programs are structured. This dissertation follows the convention introduced in [1].

II.2 Template metafunctions

The basic building block of template metaprograms is called a *template metafunction* [12, 1]. This is used as a function in template metaprogramming and is implemented as a template class. The arguments of the metafunction are the template arguments of the class, the result of the metafunction is a nested type called `type`. For example here is a template metafunction that takes a type and returns the constant version of that type. For example it turns the type `int` into `const int`.

```
template <class T>
struct make_const {
  typedef const T type;
};
```

To call the above metafunction, one should write `make_const<int>::type`. The result of this is a type, thus it can be used at all places where a type can be used.

A special case of template metafunctions is a *nullary template metafunction*. This is a metafunction that takes 0 arguments. Since it takes no arguments it is not a template class, but a class with a nested type called `type`. By providing the template arguments but not accessing the nested type called `type` of a template metafunction one gets a nullary metafunction. For example `make_const<int>` is a nullary metafunction, while `make_const<int>::type` is not.

To improve interoperability between metafunctions the following constraint is introduced: the arguments and the result of metafunctions are always types. [1]

II.3 Boxed values

Since the arguments of template metafunctions are types, to implement a metafunction operating on numbers – such as the `fact` example presented earlier – numbers need to be turned into types. A common way [1, 25, 30] of doing this is using *boxing-classes*. For example the following template class boxes integer values:

```
template <int N>
struct int_ {
  static const int value = N;
};
```

Having this template class a number, such as `13` is represented by the type `int_<13>`. This type can be passed to template metafunctions, they can do calculations with it and return other boxed values. The end result needs to be unboxed to get the result as an integer value. This unboxing happens by accessing the `value` static member of the class. For example the type `int_<13>` is unboxed by `int_<13>::value`. By convention, this static member is always called `value`.

Basic arithmetic operations on boxed values need to be implemented as template metafunctions. These metafunctions unbox the values, perform the operations and box the result again. For example the following metafunction implements multiplication:

```
template <class A, class B>
struct times {
  typedef int_<A::value * B::value> type;
};
```

The above metafunction multiplies the two arguments, `A` and `B`. It unboxes them (`A::value` and `B::value`), multiplies the unboxed values and boxes the result again.

The metafunctions operating on boxed values are implemented based on these simple arithmetic metafunctions. For example the following metafunction doubles a number:

```
template <class N>
struct double_number {
  typedef typename times<int_<2>, N>::type type;
};
```

This metafunction calls the metafunction `times` to double its argument. The `typename` keyword is used because `times<int_<2>, N>::type` is a *dependent name* [76, 31]. `double_number` calls another metafunction, `times` and returns what `times` returned. Since the result of a metafunction is its nested type called `type`, it creates a `typedef` for `times<int_<2>, N>`'s nested type called `type` in `double_number`. By publicly inheriting from `times<int_<2>, N>` `double_number` gets this nested type without having to use `typedef`. Thus, here is a simpler was of implementing `double_number`:

```
template <class N>
struct double_number : times<int_<2>, N> {};
```

This technique is commonly used during template metaprogram development, it is called *metafunction forwarding* [1].

II.4 Selection constructs

Not only integer values but other integral types, including boolean values need to be boxed as well. For example:

```
template <bool B>
struct bool_ {
  static const bool value = B;
};
```

Basic predicates, such as comparisons, logical operators, etc are implemented as template metafunctions as well. For example the following metafunction implements the `<` operator for boxed integers:

```
template <class A, class B>
struct less {
  typedef bool_<(A::value < B::value)> type;
};
```

The above code snippet unboxes its arguments, compares the unboxed values and boxes the result using a boxing template for booleans. Type selection is implemented as a metafunction based on boxed booleans:

```
template <class C, class T, class F>
struct if_;
```

It takes three arguments: a condition, `C` as a boxed boolean and two other types, `T` and `F`. The return value of this metafunction depends on the value of the condition. When it is true, the result is `T`, otherwise the result is `F`. The following code snippet implements `if_` using specialisation:

```
template <bool C, class T, class F>
struct if_c {
  typedef T type;
};

template <class T, class F>
struct if_c<false, T, F> {
  typedef F type;
};

template <class C, class T, class F>
struct if_ :
  if_c<C::value, T, F>
{};
```

A helper template class, `if_c` is used. It is similar to a metafunction but it has a non-type template argument. In Boost.MPL metafunctions with non-type template arguments get an `_c` suffix.

The Boost.MPL library [25] provides a number of boxing classes and template metafunctions (including `int_`, `bool_`, `times`, `less`, `if_` and `if_c`) that are reusable. These utilities are in the `boost::mpl` namespace, which are referred to as `mpl`, omitting `boost::` to make code examples more compact.

II.5 Higher order metafunctions

Higher order functions are functions taking functions as arguments or returning functions as the result [48, 49, 11]. This section presents the current practice [25] for providing them in template metaprogramming. Since all template metafunctions take types as arguments and return types as results, metafunctions have to be represented by types to implement higher order metafunctions. Metafunctions are template classes, which are not types. Boxing them turns them into types:

```
struct make_const {

  template <class T>
  struct apply {
    typedef const T type;
  };

};
```

The above example implements a boxed version of `make_const`. `make_const` is a class with a nested template class called `apply`. `apply` is a template metafunction – it takes a type as an argument and returns the `const` version of it as the result. This construct is called a *template metafunction class* [1]. One has to write `make_const::apply<int>::type` to call it. Metafunction classes can be passed to metafunctions as arguments or be returned by metafunctions as results. The current practice [25] is to wrap calling metafunction classes with a metafunction:

```
template <class F, class T1>
struct apply_wrap1 :
  F::template apply<T1>
{};
```

The above metafunction implements the evaluation of a metafunction taking one argument. Similar metafunctions can be written for metafunction classes taking 2, 3, etc number of arguments. These metafunctions can be automatically generated using preprocessor metaprogramming [1].

II.6 Connection to functional languages

Template metaprogramming has similarities to pure functional programming languages.

- Values are *immutable*. Every entity is defined once and its value can not be changed later.
- Template metaprogramming supports using functions as first-class citizens, thus it supports *higher-order functions*.
- Template metaprogramming supports *pattern matching* by partial template specialisation.
- Compile-time functions are *pure*. They have no side-effects and when they are evaluated with the same arguments, they return the same value.
- Template metaprogramming supports *lazy evaluation* of expressions.

It was not a design goal of C++ to support template metaprogramming and as a result of this its syntax is complicated. Non-trivial metaprograms are difficult to read and understand. When they contain bugs, it makes it difficult to find the problem and fix it. There are libraries available supporting template metaprogramming [25, 4] but they don't take advantage of the similarities between template metaprogramming and functional programming. This dissertation discusses how looking at template metaprogramming as a functional language makes it possible to develop more readable and maintainable code.

Chapter III

Functional language elements

This chapter overviews how to implement the basic building blocks of classical functional languages in C++ template metaprogramming. Metaprograms implemented using these elements are easier to read and maintain.

III.1 Laziness

The following expression assumes, that the runtime C++ functions `plus`, `minus` and `times` implementing addition, subtraction and multiplication of two numbers are provided:

```
plus(times(2, 3), minus(4, 1))
```

This expression calls the `plus` function with two arguments: `times(2, 3)` and `minus(4, 1)`. When the above expression is evaluated, each of the functions `plus`, `times` and `minus` are called once. It is guaranteed, that the arguments of `plus` – `times(2, 3)` and `minus(4, 1)` – are evaluated before `plus`, but the order in which these two expressions are evaluated is not specified. At the same time, implementing the same expression in Haskell

```
plus (times 2 3) (minus 4 1)
```

results in a different evaluation order, as Haskell starts with evaluating `plus` and evaluates its arguments only if and when they are needed.

The rules specifying in which order sub-expressions are evaluated is called the *evaluation strategy* [81] of the language. Evaluation strategies are either *strict* or *lazy* [47, 81, 49].

- Strict strategies completely evaluate the arguments passed to a function before the function call itself is evaluated.
- Non-strict strategies evaluate the arguments passed to a function only when the argument is used in the body of the function.

In template metaprogramming expressions are built from template metafunction calls, such as `f<g<int>, h<double>::type>`. The result of a template metafunction is its nested type called `type`, thus the body of the metafunction is the definition of this type. Instantiating a template class, (eg. `h` from the above example) with a type (eg. `double`) does not instantiate the nested types (eg. `h<double>::type`) until they are accessed. The evaluation of a template metafunction happens when its body is instantiated, thus when the `::type` of the metafunction is accessed. The developer has control over which `::type`s are accessed and in which order, therefore, in template metaprogramming the developer decides the evaluation order of an expression. As a result of this, the developer has to be aware of the difference between the evaluation strategies and decide in each case when to trigger the evaluation of an expression.

The evaluation strategies affect the development of template metaprograms when the developer starts combining metafunctions and writing more and more complex expressions using them. As control structures like `if_` are implemented as metafunctions, the developer of metaprograms has to make sure, that he doesn't accidentally enforce the evaluation of a branch of a selection that doesn't get selected and should not be evaluated. When an argument of a metafunction is not evaluated before the metafunction is called, the metafunction itself has to make sure that it gets evaluated at the right time if needed. Metafunctions need to be prepared for this, not all metafunctions support it. Some metafunctions accept arguments that are not evaluated yet, while others may break in such situations. The developer using the metafunctions has to know which ones support this and which don't to be able to decide the evaluation strategy he uses in his code. This affects simple expressions as well, such as the following:

```
mpl::times<
  mpl::int_<1>,
  mpl::if_<mpl::true_, mpl::int_<2>, mpl::int_<3>>
>::type
```

This example calculates 1 times 2 where the value 2 is the result of a sub-expression. However, when one tries evaluating the above expression, it breaks the compilation. `mpl::times` complains that multiplying the number `mpl::int_<1>` with an `mpl::if_<...>` value is not supported. The sub-expression `mpl::if_<...>` is passed in its original, unevaluated form to `mpl::times`. To evaluate it before `mpl::times` is called, one has to write the following:

```
mpl::times<
  mpl::int_<1>,
  mpl::if_<mpl::true_, mpl::int_<2>, mpl::int_<3>>::type
>::type
```

This code snippet has an extra `::type` after `if_<...>` enforcing its evaluation. Such sub-expressions that are passed unevaluated to functions and can be evaluated at a later point in time are called *thunk*s in a number of functional languages [48, 82]. Like in this example, the author of an expression decides the evaluation strategy by accessing or not accessing the nested `::type` of a sub-expression.

III.1.1 Laziness and recursive metafunctions

The explicit evaluation of nullary metafunctions introduces extra syntactic noise and triggers the evaluation of sub-expressions that should not be evaluated at all. For example here is an attempt to implement the factorial calculation using template metafunctions and boxed integers:

```
template <class N>
struct fact : // attempt to implement fact
  mpl::if_<
    typename mpl::less<N, mpl::int_<1>>::type,
    mpl::int_<1>,
    mpl::times<fact<mpl::minus<N, mpl::int_<1>>>, N>
  > {};
```

The above implementation uses `less` to check if the argument is less than `1`. It uses `if_` to handle arguments less than `1` in a different way than the rest of the values. Since `if_` expects a boxed boolean value and does not accept a nullary metafunction, the above implementation enforces the evaluation of `less` by explicitly accessing its nested type called `type`. When the argument is less than `1`, `fact` returns the boxed `1` value. Otherwise it evaluates `fact` with `minus<N, int_<1>>` as its argument and multiplies the result by `N`.

The problem with the above code is that the recursive call, `fact<minus<N, int_<1>>>` passes a nullary metafunction, `minus<N, int_<1>>` to `fact` instead of a boxed integer value. `fact` passes its argument to `less`, which expects a boxed integer value and does not accept a nullary metafunction, thus the above implementation breaks at the first recursive call. To fix that, one can enforce the evaluation of `minus<N, int_<1>>`:

```
template <class N>
struct fact : // attempt to implement fact
  mpl::if_<
    typename mpl::less<N, mpl::int_<1>>::type,
    mpl::int_<1>,
    mpl::times<
      fact<typename mpl::minus<N, mpl::int_<1>>::type>,
      N
    >
  >
{};
```

In this implementation the evaluation of `minus<N, int_<1>>` is enforced, thus a boxed value is given to the recursive call. However, `times` expects a boxed value as well but this version passes a thunk, `fact<typename minus<N, int_<1>>::type>` to it. To fix that, one can enforce the evaluation of this expression as well:

```
template <class N>
struct fact : // attempt to implement fact
  mpl::if_<
    typename mpl::less<N, mpl::int_<1>>::type,
    mpl::int_<1>,
    mpl::times<
      typename fact<
        typename mpl::minus<N, mpl::int_<1>>::type
      >::type,
      N
    >
  >
{};
```

The above code snippet passes a boxed value to `times` by enforcing the evaluation of `fact<typename minus<N, int_<1>>::type>`. Before the instantiation of `if_` the template arguments passed to it are evaluated. When an argument is a forced evaluation of a metafunction, such as `fact<typename minus<N, int_<1>>::type`, the forced evaluation is done and its result is used as the argument. Since in the above example it is an argument of `if_`, this evaluation happens before the instantiation of `if_`, which implements the selection logic. As a result of this, the recursive call is always evaluated before the selection could decide if it should be selected or the recursion should stop. It leads to an infinite recursion when the `fact` metafunction is evaluated. A solution for this problem is moving the recursive metafunction call into a helper metafunction:

```
template <class N> struct fact;

template <class N>
struct fact_impl :
  mpl::times<
    typename fact<
      typename mpl::minus<N, mpl::int_<1>>::type
    >::type, N>
{};

template <class N>
struct fact : // attempt to implement fact
  mpl::if_<
    typename mpl::less<N, mpl::int_<1>>::type,
    mpl::int_<1>,
    fact_impl<N>
  > {};
```

A new helper metafunction, `fact_impl` is introduced which implements the recursive part of the factorial evaluation. The forced call to `fact` is moved into this metafunction, thus the recursive call happens when `fact_impl` is evaluated. But `fact`'s implementation does not force the evaluation of `fact_impl` as it does not access its nested `::type`.

The problem with the above implementation is that for values not less than `1` it returns a nullary metafunction, `fact_impl<N>` instead of a boxed value. Boost.MPL provides the `mpl::eval_if` metafunction that takes a condition and two thunks as arguments, chooses a thunk based on the condition, evaluates it and returns the result. Using it `fact`'s implementation becomes the following:

```
template <class N> struct fact;

template <class N>
struct fact_impl :
  mpl::times<
    typename fact<
      typename mpl::minus<N, mpl::int_<1>>::type
    >::type, N>
{};

template <class N>
struct fact :
  mpl::eval_if<
    typename mpl::less<N, mpl::int_<1>>::type,
    mpl::int_<1>,
    fact_impl<N>
  >
{};
```

The above code snippet uses `eval_if` instead of `if_` to make sure that the call to the helper metafunction `fact_impl` gets evaluated when it is necessary.

III.1.2 Template metaprogramming values

Many recursive metafunctions – such as the `fact` example of the previous section – contain an `eval_if` that stops the recursion. When it should recurse, it evaluates a nullary metafunction (like `fact_impl<N>` in `fact`). When it stops the recursion, it returns some value (like `int_<1>` in `fact`). `eval_if` assumes that its second and third arguments are nullary metafunctions. The `fact` example passes `eval_if` a boxed integer, `int_<1>` as the second argument. To make sure that it works, the boxing classes (and all types that are meant to be used as values in template metaprograms) need to be turned into nullary metafunctions evaluating to themselves:

```
template <int N>
struct int_ {
  typedef int_ type;
  /*
     ...
  */
};
```

The above implementation of `int_` defines a nested type called `type` in `int_`. This nested type is a `typedef` of `int_` itself, thus `int_` is a nullary metafunction as well. These classes are *template metaprogramming value*s. A metafunction class is a value and a function at the same time - one may look at it as a value passed around in metaprograms or as a function that may be called to produce some other value. These two concepts don't interfere with each other. For example:

```
struct template_metafunction_class {
  // This makes it a metaprogramming value
  typedef template_metafunction_class type;

  // This makes it a metafunction class
  template <class A, class B>
  struct apply : /* ... */ {};
};
```

As the above example shows, a template metafunction class needs a `::type` referring to itself to be a template metaprogramming value and an `::apply` metafunction producing some result to be a higher order metafunction. Accessing `::type` will not call the higher order metafunction, it treats the class as a value. To call the function, one has to be explicit about it by using `::apply`.

A helper template class simplifies the implementation of template metaprogramming values using inheritance and the Curiously Recurring Template Pattern [76].

```
template <class T>
struct tmp_value { typedef T type; };
```

This is a template metafunction returning its argument, thus this is the identity metafunction with a different name (`tmp_value`). The implementation of `int_` becomes:

```
template <int N>
struct int_ : tmp_value<int_<N>> { static const int value; };
```

This implementation of `int_` inherits from `tmp_value` instantiated with itself to be a template metaprogramming value. While inheriting from `tmp_value` makes it explicit that the developer intended to create a template metaprogramming value, all the template arguments of `int_` have to be listed again to instantiate `tmp_value` with the right class.

There are some constraints authors of template metaprograms should keep to make writing and reading template metaprograms easier:

- *Every class used in template metaprograms as a value is a template metaprogramming value.* It guarantees, that these values work fine with lazy metafunctions. This constraint means, that the built-in types, such as `int`, `double`, etc. can not be used in template metaprograms directly, since they have no `::type` pointing to themselves. However, the following boxing class provides a wrapper for them that turns them into template metaprogramming values:

  ```
  template <class T>
  struct box { typedef box type; };

  template <class T> struct unbox;
  template <class T> struct unbox<box<T>> {typedef T type;};
  ```

  `box<int>` is a template metaprogramming value, it can be passed around in template metaprograms. The `unbox` template class implements the unboxing of such wrapped values. For example the expression `unbox<box<int>>::type` unboxes `box<int>`.

- *Every metafunction returns a template metaprogramming value.* This means that metafunctions should not return nullary metafunctions evaluating to something else. Returning a metafunction class taking some arguments to produce a value is fine, because in that case the metafunction class is a value itself. This constraint guarantees that using `::type` is always a safe operation, since it can not accidentally change an already evaluated value. To evaluate a template metafunction class and get some other value, one has to use its `::apply<...>` template.

Template metafunctions not accepting nullary metafunctions as arguments make the implementation of template metaprograms complicated. Given that types that are used as values in template metaprogramming – such as boxing classes – are expected to work as nullary metafunctions evaluating to themselves, template metafunctions should be implemented in a way that they accept nullary metafunctions as well. Here is an implementation of `times` that accepts nullary metafunctions as arguments and evaluates them:

```
template <class A, class B>
struct times {
  typedef mpl::int_<A::type::value * B::type::value> type;
};
```

The above implementation of `times` assumes, that its arguments are nullary metafunctions evaluating to boxed integers. It evaluates them first (`A::type` and `B::type`), unboxes the results (`A::type::value` and `B::type::value`), multiplies them and boxes it again. A metafunction that works with nullary metafunctions as well is called a *lazy metafunction* [1]. When metafunctions are lazy, the implementation of `fact` from section III.1.1 becomes significantly simpler:

```
template <class N>
struct fact :
  eval_if<
    less<N, mpl::int_<1>>,
    mpl::int_<1>,
    times<fact<minus<N, mpl::int_<1>>>, N>
  > {};
```

The above implementation builds on the fact that all metafunctions it calls – `eval_if`, `less`, `times`, `fact`, `minus` – are lazy. As a result of this, the explicit usage of `typename ... ::type` can be omitted and no helper metafunctions are needed making the code more compact.

III.1.3 Changing the evaluation strategy of expressions

Expressions in template metaprogramming – such as `times<fact<minus<N, int_<1>>>, N>` – are built of template metafunction calls. They are called *angle bracket expression*s. The evaluation order depends on the expression itself.

- `times<fact<minus<N, int_<1>>>, N>::type` evaluates the expression lazily by passing `fact<minus<N, int_<1>>>` and `N` to `times` as arguments.
- `times<fact<minus<N::type, int_<1>>::type>::type, N::type>::type` evaluates the expression strictly: all arguments are evaluated before they are passed to the metafunctions called from the expression.

The evaluation strategy depends on the expression itself. This section presents how to simulate lazy evaluation for an expression calling metafunctions that are not prepared for lazy evaluation. The metafunctions called from the expression don't need to be changed for this. This simulation is implemented by a template class, `lazy`. Here is an implementation of `fact` using it:

```
template <class N>
struct fact :
  mpl::eval_if<
    typename mpl::less<N, mpl::int_<1>>::type,
    mpl::int_<1>,
    lazy<mpl::times<fact<mpl::minus<N, mpl::int_<1>>>, N>>
  >
{};
```

In this example the template class `lazy` is used to evaluate the calls to the `times`, `fact` and `minus` non-lazy template metafunctions in a lazy way. Such a tool makes it possible to add laziness to an existing metaprogramming library that does not support laziness without having to change the library.

Implementing `lazy`

`lazy` is a metafunction taking a nullary metafunction as argument. `lazy` evaluates its argument when `lazy` itself is evaluated. When an angle bracket expression is passed to `lazy`, it evaluates a transformed version of the expression: all arguments the expression passes to a metafunction get evaluated and the results are passed to the metafunction. Thus when a nullary metafunction is passed as an argument to another metafunction, the nullary metafunction gets evaluated and the result is passed to the other metafunction. Here is an implementation of `lazy`:

```
template <class Exp>
struct lazy : Exp {};
```

This implements how `lazy` should deal with expressions that can not be transformed. By inheriting from `Exp`, `lazy` guarantees that when `lazy` is evaluated, `Exp` is also evaluated.

Partial specialisation can be used to detect and handle angle bracket expressions. These specialisations can be implemented by template template arguments. A partial specialisation can match the top-level metafunction call of the expression. For example, when the top-level metafunction call passes three arguments to a metafunction, the following specialisation implements the lazy evaluation of the expression:

```
template <
  template <class, class, class> class T,
  class A1, class A2, class A3
>
struct lazy<T<A1, A2, A3>> :
  T<
    typename lazy<A1>::type,
    typename lazy<A2>::type,
    typename lazy<A3>::type
  >
{};
```

It defines a partial specialisation of `lazy` for some `T` template class taking 3 arguments and some `A1`, `A2` and `A3` classes passed as arguments to this template class. `lazy` evaluates the arguments following this lazy evaluation logic by evaluating `lazy<A1>`, `lazy<A2>` and `lazy<A3>`. The results of these evaluations are passed to `T` as arguments. All these evaluations happen only when `lazy` itself is evaluated.

This implements the lazy evaluation strategy for angle bracket expressions, where the top-level metafunction call is a call to a metafunction with three arguments. Similar specialisations can be implemented for top-level metafunction calls with different arity. The Boost.Preprocessor library [34] provides tools to generate these specialisations automatically.

Protecting metafunctions that are already lazy

`lazy` works for metafunctions that are not lazy, but in some cases it causes issues for metafunctions that are lazy. Lazy metafunctions may need that their arguments are not evaluated. For example the template metafunction `eval_if` takes a condition as its first argument and two nullary template metafunctions as its second and third arguments. It chooses exactly one of the two nullary metafunctions based on the condition and evaluates only that one. The other nullary metafunction remains unevaluated. In several cases evaluating the other nullary metafunction than the selected one leads to a compilation error and should be avoided.

Tools like `lazy` evaluate all arguments of a template metafunction call before making the metafunction call, thus for `eval_if` they evaluate both sides of the selection, not just the selected one. This should be avoided. A marker can tell `lazy` to stop recursing. This dissertation implements it using a template class, `already_lazy`. The following implementation of the `fact` function presented in section III.1.1 uses it:

```
template <class N>
struct fact : // attempt to implement fact
  lazy<
    eval_if<
      less<N, int_<1>>,
      already_lazy<int_<1>>,
      already_lazy<times<fact<minus<N, int_<1>>>, N>>
    >> {};
```

The two nullary metafunctions `eval_if` has to choose from are wrapped with `already_lazy` to protect them from the forced evaluation. The `lazy` template class needs to be specialised the following way:

```
template <class T> struct already_lazy;

template <class Expr>
struct lazy<already_lazy<Expr>> { typedef Expr type; };
```

This handles the parts of the expressions that are protected by `already_lazy`. It leaves the content wrapped by this template class untouched.

`already_lazy` protects against evaluating the arguments of lazy metafunctions used in an expression transformed by `lazy`, but it also guarantees that the wrapped expression is left untouched, thus it will not be evaluated following `lazy`'s logic. This breaks the code in many cases, even in the above `fact` example, where the `times<fact<minus<N, int_<1>>>, N>` expression – protected by `already_lazy` – needs to be evaluated following `lazy`'s evaluation strategy. Using `lazy` inside `already_lazy` resolves this:

```
template <class N>
struct fact :
  lazy<
    eval_if<
      less<N, int_<1>>,
      already_lazy<int_<1>>,
      already_lazy<lazy<times<fact<minus<N, int_<1>>>, N>>>
    >> {};
```

The nested `lazy` inside `already_lazy` guarantees that the `times<fact<minus <N, int_<1>>>, N>` expression is evaluated following `lazy`'s evaluation strategy. Wrapping `eval_if`'s arguments with `already_lazy` guarantees that it does not get evaluated too early, using `lazy` inside `already_lazy` guarantees that the recursion of `lazy` continues, but only when `eval_if` triggers the evaluation of that expression.

Making it convenient to pass lazy arguments to metafunctions

The following template class wraps the pattern used in the above example – using `lazy` inside `already_lazy`:

```
template <class Expr>
struct lazy_argument;

template <class Expr>
struct lazy<lazy_argument<Expr>> {
  typedef lazy<Expr> type;
};
```

The above code snippet specialises `lazy` for `lazy_argument` elements. When a sub-expression is wrapped with it, `lazy` returns the lazy version of the expression as a nullary metafunction, thus it can be evaluated later. Template aliases provided by C++11 simplify the implementation of `lazy_argument`:

```
template <class Expr>
using lazy_argument = already_lazy<lazy<Expr>>;
```

This implementation makes use of the fact that `lazy_argument` can be expressed by the combination of `already_lazy` and `lazy`. Here is an implementation of `fact` using `lazy_argument`:

```
template <class N>
struct fact :
  lazy<
    eval_if<
      less<N, int_<1>>,
      lazy_argument<int_<1>>,
      lazy_argument<times<fact<minus<N, int_<1>>>, N>>
    >
  >
{};
```

This code snippet uses `lazy_argument` to protect the two branches of `eval_if` from being evaluated too early and ensures that when they are evaluated, they are evaluated lazily.

Arguments of functions implemented using `lazy`

There is another thing users of tools like `lazy` need to be careful with. When `fact` is called with an angle bracket expression as its argument, its template argument, `N` refers to that angle bracket expression. For example when

```
fact<
  eval_if<true_, int_<1>, divides<int_<1>, int_<0>>>
>::type
```

is evaluated, `N` refers to `eval_if<true_,int_<1>,divides<int_<1>,int_<0>>>`. Using `N::type` evaluates that expression, but when `N` is referred to inside `lazy`, the way `N` is evaluated is changed by `lazy`. The evaluation strategy used inside `fact` should be an implementation detail, but this behaviour affects the callers of the function. Because of wrapping `eval_if<...>` with `lazy` without using `lazy_argument` the above example would break the compilation.

To avoid this, references to metafunction arguments should be wrapped by `already_lazy` inside `lazy` blocks. The following implementation of `fact` demonstrates this:

```
template <class N>
struct fact :
  lazy<
    eval_if<
      less<already_lazy<typename N::type>, int_<1>>,
      lazy_argument<int_<1>>,
      lazy_argument<
        times<
          fact<minus<already_lazy<typename N::type>, int_<1>>>,
          already_lazy<typename N::type>
        >
      >
    >
  >
{};
```

This example evaluates its arguments to make sure that the metafunction is lazy, but wraps this evaluated argument with `already_lazy` to avoid `lazy` changing the meaning of the arguments. Using this implementation makes it work when an unevaluated expression, such as `eval_if<true_, int_<1>, divides<int_<1>, int_<0>>>` is given to `fact` as its argument, since `already_lazy`, that wraps `N` protects it from being evaluated in a different way the calling code intended to evaluate it.

Protecting all arguments of a metafunction call

The above example works, however, the argument `N` is evaluated in a strict way by explicitly accessing its `::type`. Another template class is needed that supports situations where a sub-expression, such as `N` has to be evaluated but `lazy` should not recurse into it. This means, that when the sub-expression is a metafunction call, its arguments should not be transformed by `lazy`. The following code snippet implements the `lazy_protect_args` template class providing this:

```
template <class Expr>
struct lazy_protect_args;

template <class Expr>
struct lazy<lazy_protect_args<Expr>> : Expr {};
```

The above code snippet defines the template class `lazy_protect_args`. When a sub-expression is wrapped with this, `lazy` evaluates the expression but does not change it. Here is a completely lazy implementation of `fact` using `lazy_protect_args`:

```
template <class N>
struct fact :
  lazy<
    eval_if<
      less<lazy_protect_args<N>, int_<1>>,
      lazy_argument<int_<1>>,
      lazy_argument<
        times<
          fact<minus<lazy_protect_args<N>, int_<1>>>,
          lazy_protect_args<N>
        >
      >
    >
  >
{};
```

This version of `fact` uses `lazy_protect_args` to ensure that `N` is evaluated and the evaluation happens following the evaluation strategy the caller of `fact` intended to use.

Summary

This section has introduced a number of utilities changing the evaluation strategy of an expression:

- `lazy<Exp>` changes the way `Exp` is evaluated. It guarantees that inside the expression `Exp` the arguments of every metafunction are evaluated *before* the metafunction is called. It only affects the expression itself, it does not affect metafunctions called by this expression.
- `already_lazy<Exp>` used inside `lazy`'s `Exp` protects a sub-expression from being transformed. The evaluation strategy of sub-expressions wrapped by `already_lazy` are not changed.
- `lazy_argument<Exp>` used inside `lazy`'s `Exp` protects the arguments of a lazy metafunction. It protects the expression it wraps from being evaluated by `lazy`, so the function it is passed to as an argument can evaluate it at a later point in time. When that evaluation happens, the expression `lazy_argument` wraps is evaluated following `lazy`'s evaluation strategy.
- `lazy_protect_args<Exp>` used inside `lazy`'s `Exp` protects the evaluation of the lazy metafunctions of an expression inside `lazy`. It guarantees that the arguments of the template metafunction call it wraps are not transformed by `lazy`.

This section has presented how an angle bracket expression can be evaluated lazily when the metafunctions being called from it don't support it. The approach presented in this section does not require any changes to the implementation of these metafunctions. This section has also discussed a number of issues that arise when someone uses this approach and ways of avoiding them.

III.2 Currying

Currying [48] is supported by several functional languages [48, 50, 46]. The idea behind it is by applying one argument to a function expecting `n` arguments one gets a function expecting `n - 1` arguments. By applying a second argument to this new function one gets another function expecting `n - 2` arguments. This is repeated until `n` arguments have been provided and these arguments are collected. When the last, the `n`th argument is provided, the original function is evaluated.

For example providing only one argument, `int_<2>` to the `times` function one gets a function expecting one argument. This new function multiplies its argument by `2` – thus currying simplifies the implementation of `double_number` presented earlier. The following code snippet implements the curried version of `times`:

```
template <class A>
struct times : tmp_value<times<A>> {
  template <class B>
  struct apply : int_<A::value * B::value> {};
};
```

The above implementation of `times` takes one argument and returns a metafunction class taking another argument. The result of calling the metafunction class is the multiplication of the two arguments. The above code snippet implements currying manually. Doing it this way is error prone and makes the code unreadable, since it forces the developer to write a large amount of boilerplate code.

This section presents how to implement metafunctions that can turn non-curried metafunctions into a curried ones. The metafunctions doing these transformations are called `curry1`, `curry2`, etc. The following example presents how to generate the curried version of `times`, `curried_times` from the non-curried one (`times`):

```
typedef curry2<times> curried_times;
```

The above code snippet uses a template class, `curry2` to turn the template metafunction `times` into a curried one. `curry2` transforms metafunctions taking two arguments. Different versions, such as `curry3`, `curry4`, etc can be implemented for different metafunction arities.

`curryN` is a template metafunction class taking its arguments one by one and collecting them in a typelist [3]. When all of the arguments are provided, it passes them to the wrapped template metafunction. In order to do that the original metafunction – eg. `times` – has to be turned into a metafunction that accepts a typelist as its argument list. For example a metafunction taking one argument, a typelist of size 2 needs to be generated from the metafunction `times` taking two arguments. Boost.MPL [25] provides tools for that. The following example assumes, that a metafunction called `times` taking two boxed numbers and multiplying them is available.

```
mpl::unpack_args<mpl::quote2<times>>
```

`mpl::quote2` turns a metafunction taking 2 arguments into a metafunction class. `mpl::unpack_args` turns a metafunction class taking multiple arguments into a metafunction class taking one argument, a typelist which is treated as the argument list of the original function. The following code snippet presents how to use the `deque` compile-time container implementation of Boost.MPL to call the metafunction class the above expression produces:

```
mpl::apply_wrap1<
  mpl::unpack_args<mpl::quote2<times>>,
  mpl::deque<mpl::int_<2>, mpl::int_<3>>
>
```

The above code snippet evaluates `times<mpl::int_<2>, mpl::int_<3>>` by passing a typelist with two elements, `mpl::int_<2>` and `mpl::int_<3>` to the metafunction class generated above.

This section uses a helper metafunction, `curry_impl` to implement `curry1`, `curry2`, etc. `curry_impl` collects the arguments one by one for currying. It takes three arguments: the unpacked version of the original metafunction, the number of missing arguments and the typelist of already collected arguments. The implementation of `curry1`, `curry2`, etc uses `curry_impl`. For example:

```
template <template <class, class> class F>
struct curry2 :
  curry_impl<
    mpl::unpack_args<mpl::quote2<F>>,
    mpl::int_<2>,
    mpl::deque<>
  >
{};
```

`curry2` takes the metafunction to curry as a template template argument. It builds the unpacked version of it using `mpl::unpack_args` and `mpl::quote2` and passes it to `curry_impl`. Since initially no arguments are available, it passes the number `2` and an empty typelist, `mpl::deque<>` to `curry_impl`. Writing the `curryN` functions manually is repetitive and error prone, but the Boost.Preprocessor library [34] provides tools to automatically generate them. The details of this are not presented here.

Here is a recursive implementation of `curry_impl`. It's implementation is separated into two metafunctions, `curry_impl` and `next_currying_step` due to the lack of laziness in Boost.MPL:

```
template <class UnpackedF, class ArgNumLeft, class ArgsSoFar>
struct curry_impl :
  mpl::eval_if<
    typename mpl::equal_to<ArgNumLeft, mpl::int_<0>>::type,
    mpl::apply_wrap1<UnpackedF, ArgsSoFar>,
    next_currying_step<UnpackedF, ArgNumLeft, ArgsSoFar>
  >
{};
```

The above implementation checks if there are more arguments to collect by checking if `ArgNumLeft` is already `0`. If there are no further arguments required, the original metafunction is evaluated by passing the collected arguments, `ArgsSoFar` to the unpacked version of the original metafunction. This is implemented by `mpl::apply_wrap1<UnpackedF, ArgsSoFar>`. When there are further arguments to collect, a metafunction class taking 1 argument is returned. This is implemented by `next_currying_step`:

```
template <class UnpackedF, class ArgNumLeft, class ArgsSoFar>
struct next_currying_step {
  typedef next_currying_step type;

  template <class T>
  struct apply :
    curry_impl<
      UnpackedF,
      typename mpl::minus<ArgNumLeft, mpl::int_<1>>::type,
      typename mpl::push_back<ArgsSoFar, T>::type
    >
  {};
};
```

The above implementation returns a template metafunction class taking one argument, appending this argument to the collected argument list by `mpl::push_back<ArgsSoFar, T>`, reducing the number of missing arguments by `mpl::minus<ArgNumLeft, mpl::int_<1>>` and passing these to the metafunction `curry_impl`.

The above technique automates the generation of curried template metafunctions. It adds currying to any metafunction without having to change the original one. It can be used to add currying to third party metafunctions as well.

III.3 Algebraic data types

Algebraic data types are basic language elements in a number of functional languages, such as Haskell, ML, Clean etc. This section describes a method of representing them in C++ template metaprogramming – a functional language not prepared for algebraic data types explicitly. Algebraic data types in Haskell have the following form:

```
data <name> [<type arguments>] =
  <constructor name> <constructor arguments> |
  <constructor name> <constructor arguments> |
  ...
```

Here is an implementation of the `Maybe` type:

```
data Maybe a = Just a | Nothing
```

Values of type `Maybe a` are either values of type `a` or a special value, `Nothing`. Two constructors are provided for constructing these values:

- `Just a` for constructing `Maybe a` values representing values of type `a`.
- `Nothing` for constructing the special `Nothing` value.

`Maybe` can be used for error reporting [69]: a function producing a value of type `a` returning a `Maybe a` value makes it possible to report failures. `Just a` represents success and the value of type `a` it wraps is the result. `Nothing` represents failure. This is a simple and limited way of error handling, since there is no way for a function to return a detailed error message helping the user to find the root cause.

This dissertation implements each constructor by a C++ template. The constructor arguments are the template arguments. The following example implements the constructor `Just a`:

```
template <class A>
struct just : tmp_value<just<A>> {};
```

Note that the Haskell implementation of `Just` has an argument of type `a` while the argument of the C++ template metaprogramming version is a `class`. In the Haskell code `a` is a type variable referring to a concrete type. `Maybe` is a *type constructor* [48], `a` is its type parameter. For example `Maybe Int` is a type created using this type constructor and `a` refers to `Int`, therefore the `Just` constructor of this type has an argument of type `Int`. In

C++ template metaprograms the arguments of metafunctions are always `class`es. This approach can not represent types, thus the type information is lost during the approach presented here for implementing `Maybe` in template metaprogramming. Algebraic data types and their arguments have no direct representation in C++ template metaprogramming, only the constructors are implemented.

As constructors are used to construct template metaprogramming values, by using `tmp_value`, `just<...>::type` gives `just<...>`. The following code snippet implements the other constructor of the `Maybe` type, `Nothing`:

```
struct nothing : tmp_value<nothing> {};
```

Since `Nothing` has no arguments it is represented by a class instead of a template class. The approach presented in this dissertation supports the implementation of the constructors of algebraic data types in C++ template metaprogramming, but it doesn't not support representing the connection between them.

III.3.1 Laziness

One way of looking at the constructors of algebraic data types is that they are functions returning a value. For example `just` is a template metafunction taking the value to wrap with `Just` and it returns the wrapped value. To wrap the result of `mpl::plus<mpl::int_<8>, mpl::int_<5>>`, one would write:

```
just<mpl::plus<mpl::int_<8>, mpl::int_<5>>>::type
```

The above code snippet uses `just` as a metafunction to produce the wrapped version of the result of an addition. It uses `just` as a lazy metafunction, since it passes `just` the unevaluated expression `mpl::plus<mpl::int_<8>, mpl::int_<5>>` and expects `just` to evaluate it.

The constructors of algebraic data types should be turned into lazy template metafunctions. For example in the above expression, `just<mpl::plus< mpl::int_<8>,mpl::int_<5>>>::type` refers to the following:

```
just<mpl::plus<mpl::int_<8>, mpl::int_<5>>::type>
```

The `::type` of the constructor of an algebraic data type should evaluate its arguments and instantiate the same constructor with the evaluated arguments. Here is an implementation of `just` based on this:

```
template <class A>
struct just { typedef just<typename A::type> type; };
```

`type` is a `typedef` of `just` instantiated with the evaluated argument, `A::type`. Implementing all constructors of algebraic data types this way produces a large amount of boilerplate code and is error prone. It is easy to forget evaluating arguments. However, the Boost.Preprocessor library [34] provides tools for wrapping the construction of algebraic data types with macros. The details of this wrapping are not presented here, a reference implementation can be found at [59].

III.3.2 Currying

As one can look at the constructors of algebraic data types as functions, they should support currying as well. Let's introduce a new algebraic data type that can be used as an example for currying. This algebraic data type represents lists:

```
template <class Head, class Tail> struct cons;
struct empty;
```

The empty list is represented by `empty`, `cons<Head, Tail>` represents a list with `Head` as its first element and `Tail` as the list of remaining elements. The following example represents the list `[1, 2, 3]`:

```
cons<
  mpl::int_<1>,
  cons<
    mpl::int_<2>,
    cons<
      mpl::int_<3>,
      empty
    >
  >
>
```

This code snippet uses `cons` recursively to construct a list with three elements. The `cons` constructor has two arguments. To support currying, giving it only one should return a metafunction class expecting the remaining list and producing a `cons` value. For example:

```
typedef cons<mpl::int_<1>>::type append_to_1;
```

The above code snippet defines the metafunction class `append_to_1` by passing only one argument to `cons`, the head of the list which is `mpl::int_<1>`. It produces a metafunction class that expects a list as argument and appends it to the list `[1]`. The following example uses it to build the list `[1, 2, 3]`:

```
append_to_1::apply<
  cons<
    mpl::int_<2>,
    cons<
      mpl::int_<3>,
      empty
    >
  >
>::type
```

This code snippet calls `append_to_1` with the `[2, 3]` list as argument and produces the `[1, 2, 3]` list.

Currying has to be implemented for every constructor individually, however, it can be automatically generated by the same macro mentioned in the previous section. The implementation of it is not presented here. A reference implementation can be found at [59].

III.4 Typeclasses

The Haskell language provides *typeclasses* [48] for implementing function overloading. There is a known similarity between Haskell typeclasses and C++ concepts [8, 21, 22, 24, 23, 73, 71], however, concepts are not part of C++ at the time of writing this. This section presents a solution for implementing typeclasses in conformity with the C++ standard.

A typeclass defines an interface for a type. It takes the type as argument and declares a number of functions using that type in their signature. The following example shows the syntax of creating a typeclass:

```
class EqualityComparable a where
  equal :: a -> a -> Bool
  notEqual :: a -> a -> Bool
```

This example defines a typeclass called `EqualityComparable`. Its argument is called `a` and two functions are specified: `equal` and `notEqual`. Both of them take two values of type `a` as arguments and return a boolean value.

Types can be instances of a typeclass. Every type has to be explicitly made an instance of a typeclass by implementing the expected functions. The following example shows the syntax of making a type an instance of a typeclass:

```
instance EqualityComparable Int where
  equal x y = x == y
  notEqual x y = x /= y
```

This example uses the comparison operators for implementing the two functions. Certain functions required by a typeclass can have default implementations. Instances can override this default, but every instance not overriding it inherits the default version. The following example shows the syntax of providing a default implementation for a function:

```
class EqualityComparable a where
  equal :: a -> a -> Bool
  notEqual :: a -> a -> Bool
  notEqual x y = not (equal x y)
```

This example uses `equal` to implement `notEqual`. This dissertation presents an implementation of typeclasses in C++ template metaprogramming based on the idea of *traits* [42]. A trait is a template class with member types and static member constants. This template class is specialised for different types as template arguments and define the nested types and static member constants differently for every template argument. It is used to encode extra information about types that can be consumed by template metaprograms.

A typeclass is implemented as a trait. The argument of the typeclass is the template argument of the trait. This approach does not support the explicit representation of the list of expected functions. The following example shows the `EqualityComparable` typeclass implemented in template metaprogramming:

```
template <class A>
struct equality_comparable;
```

This template class has no implementation. This ensures that when it is used inappropriately, the compiler emits an error message at the place of misuse and the user doesn't get a confusing error message at a later point in the compilation process.

Boost.MPL [25] uses *tag*s to implement template metafunction overloading [1]. A tag is a class that is used as an identifier in template metaprogramming. Boost.MPL uses tags as dynamic type information. This implementation of typeclasses expects tags as template arguments. Specialising the template class of a tag and implementing the expected functions as template metafunction classes – classes with a nested metafunction called `apply` [1] – makes a tag an instance of that typeclass. The following example shows how to make the boxed integers of Boost.MPL instances of the example typeclass defined above:

```
template <>
struct equality_comparable<integral_c_tag> {
  struct equal : tmp_value<equal> {
    template <class A, class B>
    struct apply : mpl::equal_to<A, B> {};
  };
  struct not_equal : tmp_value<not_equal> {
    template <class A, class B>
    struct apply : mpl::not_equal_to<A, B> {};
  };
};
```

This code snippet specialises the `equality_comparable` template class for the type `integral_c_tag` and implements the two expected operations, `equal` and `not_equal`, as metafunction classes. These implementations use comparison functions provided by Boost.MPL.

Functions related to a typeclass can be called using the traits representing the typeclass. Unfortunately the calling code has to specify the tag explicitly. The following example implements a function, `self_equal`, using the `equality_comparable` typeclass in both languages:

```
-- Haskell
selfEqual :: EqualityComparable a => a -> Bool
selfEqual x = equal x x

// Template metaprogramming
template <class X> struct self_equal : apply<
  typename equality_comparable<
    typename mpl::tag<X>::type
  >::equal,
  X
> {};
```

The requirement, that the argument `x` has to be an instance of a typeclass is encoded in a different way in the two languages. In Haskell it is encoded in the type of the function by having an expectation on the type argument, while in template metaprogramming it is encoded in the implementation of the function by accessing an element of the trait.

This dissertation implements expected functions with default implementations in template metaprogramming by creating a second template class for the typeclass containing the default implementations as metafunction classes. Every instance of the typeclass has to instantiate this extra template class and inherit publicly from the instance. Here is an example for the extra template class and the updated instance:

```
template <class A>
struct equality_comparable_defaults {

  struct not_equal : tmp_value<not_equal> {
    template <class A, class B>
    struct apply :
      mpl::not_<
        typename mpl::apply<
          typename equality_comparable<A>::equal,
          A,
          B
        >::type
      >
    {};
  };

};

template <>
struct equality_comparable<integral_c_tag> :
equality_comparable_defaults<integral_c_tag> {
  // not_equal is inherited
  struct equal { /* Same as before... */ };
};
```

The default implementation of `not_equal` uses the `equal` method of the `equality_comparable` typeclass. Instances can override this default implementation by overriding the nested class. Using this approach to implement typeclasses in template metaprogramming has several advantages.

- It helps *structuring the code* by grouping the functions implementing the same abstract concept – what the typeclass represents – in one class.
- Given the fact that typeclasses are always used explicitly, it helps the compiler to provide *meaningful error messages*, since the name of the typeclass is likely to appear in the error messages when a tag is not an instance of the typeclass the code is trying to use.

This approach has drawbacks as well.

- It doesn't support specifying the list of expected functions. The author of a typeclass can express it using comments or in the documentation, but not in a way that the compiler understands.
- The compiler can not verify and enforce the existence and the expected signature of the required functions. Error messages are generated the first time a missing function is called.

In spite of the drawbacks, following this approach helps making template metaprograms more structured.

III.5 Angle-bracket expressions as first class citizens

This section presents how to treat angle-bracket expressions as first class citizens in template metaprogramming and how to implement the basic operations for them.

III.5.1 Syntaxes

In template metaprogramming, expressions are implemented by angle bracket expressions. Being able to pass them around in template metaprograms makes it possible to implement complex control structures. But angle bracket expressions are not template metaprogramming values - when someone tries accessing their `::type` they get evaluated. They could be boxed as any other type, but the only thing that can be done with a boxed type is unboxing it, while there are more operations for angle bracket expressions. The following code snippet provides a wrapper for angle bracket expressions:

```
template <class T>
struct syntax { typedef syntax T; };
```

This works the same way as `box`, however, instead of providing a metafunction for unwrapping them, the following metafunction, `eval_syntax` is provided for unwrapping and evaluating a syntax:

```
template <class T> struct eval_syntax;
template <class T> struct eval_syntax<syntax<T>> : T {};
```

By inheriting from the wrapped expression, the `::type` of `eval_syntax` is the result of evaluating the wrapped angle bracket expression.

For example `syntax<mpl::plus<mpl::int_<11>, mpl::int_<2>>>` represents the expression `11+2`. Accessing `syntax<mpl::plus<mpl::int_<11>, mpl::int_<2>>>::type` gives the syntax back, thus it can not be accidentally evaluated. To get the value `13`, one has to evaluate `eval_syntax<syntax< mpl::plus<mpl::int_<11>, mpl::int_<2>>>>::type` which unwraps and then evaluates the syntax.

III.5.2 Variables

Syntaxes may have placeholders inside the angle bracket expression. These placeholders can later be replaced by sub-expressions. This dissertation refers to such placeholders as *variable*s. To be able to differentiate them, such variables need identifiers. The approach presented here uses types as the identifiers of variables. Variables are represented by instances of a template class:

```
template <class Id>
struct var : tmp_value<var<Id>> {};
```

Variables are the instances of the above `var` template class. The `Id` type argument is the identifier of the variable. Since variables are template metaprogramming values, they can be passed around in template metaprograms as other values. To create a variable, one should create an identifier for it and then instantiate `var` using that identifier:

```
struct x_;

var<x_> // this is the variable
```

The class `x_` is declared to be the identifier of the variable (any class that has been declared can be used as an identifier) and `var` is instantiated with it. The expression `11+x_` is represented by `syntax<mpl::plus<mpl::int_<11>, var<x_>>>`. As a syntactic sugar, one may define `typedef var<x_> x;` to

simplify the code using the variable. Now `x` refers to the variable `x_` and the above expression becomes `syntax<mpl::plus<mpl::int_<11>, x>>`. The rest of this dissertation refers to such variables by one character long, lower case identifiers, such as `a`, `b`, `c`, `x`, `y`, etc.

III.5.3 Let expressions

Syntaxes with variables become useful when *variable substitution* is implemented. Occurrences of a variable in a syntax should be replaced with a syntax. All occurrences should be replaced with the same syntax. For example given the following syntax:

```
syntax<mpl::times<mpl::plus<x, y>, x>>
```

Replacing occurrences of `x` with `syntax<mpl::int_<13>>` gives

```
syntax<mpl::times<mpl::plus<mpl::int_<13>, y>, mpl::int_<13>>>
```

All occurrences of `x` are replaced with `mpl::int_<13>`, but `y` remained unchanged. When the expression a variable is replaced with contains no variables, it is a thunk that may be evaluated at any point in time. Substituting a variable with a thunk is similar to *let expressions* [48, 70] of many functional languages, that can be used to bind values to names. For example a simple let expression in Haskell looks like the following [48]:

```
let
  x = f 11
in
  x + x
```

This example binds the expression `f 11` to `x`. Based on this commonality, the metafunction implementing variable substitution is called `let`. The following example shows how `let` can be used. It replaces occurrences of the variable `x` with the syntax `f<mpl::int_<11>>` in the syntax `syntax<mpl::plus<x, x>>`.

```
let<
  x, syntax<f<mpl::int_<11>>>,
// in
  syntax<mpl::plus<x, x>>
>::type
```

In this example the comment `// in` is added to make it more like the Haskell example above. This code snippet declares `let`:

```
template <class A, class E1, class E2>
struct let;
```

It takes three arguments:

- `A`, the variable to substitute.
- `E1`, the syntax to replace occurrences of `A` with.
- `E2`, the syntax to do the substitution in.

`let` does the substitution and returns the substituted syntax. The above example gives

```
syntax<mpl::plus<f<mpl::int_<11>>, f<mpl::int_<11>>>>
```

As syntaxes are template metaprogramming values, passing them to and returning them from `let` does not cause issues. `let` is a metafunction transforming syntaxes.

`let` substitutes all occurrences of one variable. When an expression has more than one variables, `let` needs to be used repeatedly to substitute them all. For example:

```
let<
  x, syntax<f<mpl::int_<11>>>,
  syntax<
    let<
      y, syntax<g<mpl::int_<13>>>,
      syntax<mpl::plus<x, y>>
    >::type
  >
>::type
```

This expression takes the syntax `syntax<mpl::plus<x, y>>`, substitutes `y` with `syntax<g<mpl::int_<13>>>` and `x` with `syntax<f<mpl::int_<11>>>` after that. The result of this is `syntax<mpl::plus<f<mpl::int_<11>>, f<mpl::int_<13>>>>`.

Having to write `syntax` explicitly everywhere, where `let` is used makes the code more difficult to read. Using `syntax` is important to protect the wrapped angle-bracket expressions from being accidentally evaluated, but it introduces a large amount of syntactic noise in the resulting code. The following template class simplifies code using let expressions:

```
template <class A, class E1, class E2>
struct let_c : let<A, syntax<E1>, syntax<E2>> {};
```

This template class, `let_c` takes the same arguments as `let`, but it takes angle-bracket expressions as its second and third arguments. All it does is wrapping `E1` and `E2` with `syntax` and passing them to `let`. It guarantees, that `E1` and `E2` are not evaluated and calls the safe `let` metafunction to process them. The name, `let_c` follows the naming convention of Boost.MPL, which provides `_c` versions of metafunctions accepting unboxed scalar values as arguments. `let_c` accepts unboxed angle-bracket expressions. Here is a simplified version of the above example using `let_c`:

```
let_c<
  x, f<mpl::int_<11>>,
  let_c<
    y, g<mpl::int_<13>>,
    mpl::plus<x, y>
  >
>::type
```

This example is the same as the one on page 48, but it uses `let_c` instead of `let` to make the code easier to read. This code snippet returns a syntax. To evaluate it, one has to use `eval_syntax`. Thus, to get the result of `f(11) + g(13)` one has to use:

```
eval_syntax<
  let_c<
    x, f<mpl::int_<11>>,
    let_c<
      y, g<mpl::int_<13>>,
      mpl::plus<x, y>
    >
  >
>::type
```

This code snippet does the substitution of `x` and `y` by using `let_c` and evaluates the result by using `eval_syntax`. The following code snippet defines `eval_let` to simplify code combining `eval_syntax` and `let`:

```
template <class A, class E1, class E2>
struct eval_let : eval_syntax<let<A, E1, E2>> {};
```

This metafunction has the same arguments as `let`, it calls `let` with these arguments and evaluates the resulting syntax immediately. `eval_let_c` can be implemented in a similar way. Here is a simplified version of the above example using it:

```
eval_let_c<
  x, f<mpl::int_<11>>,
  let_c<
    y, g<mpl::int_<13>>,
    mpl::plus<x, y>
  >>::type
```

This code snippet uses `eval_let_c` instead of the combination of `eval_syntax` and `let_c` to keep the code simple. It replaces only the outer `let_c` with `eval_let_c` to avoid evaluating the `mpl::plus<...>` expression when only `y` has been substituted.

The implementation of `let`

One can look at `let` as a language element, but it is implemented as a metafunction as all of its arguments are template metaprogramming values. `let` evaluates its arguments and passes them to `strict_let`, which is a helper metafunction:

```
template <class A, class E1, class E2>
struct let :
  strict_let<
    typename A::type,
    typename E1::type,
    typename E2::type
  > {};
```

This code snippet evaluates the arguments to ensure laziness of `let` and passes them to `strict_let` doing the substitution. `strict_let` uses partial specialisation to unwrap the syntaxes:

```
template <class A, class E1, class E2>
struct strict_let;

template <class A, class E1, class E2>
struct strict_let<A, syntax<E1>, syntax<E2>> :
  syntax<typename let_in_syntax<A, E1, E2>::type> {};
```

strict_let unwraps the two syntaxes using partial template specialisation and instantiates the template class let_in_syntax using them. This template takes and returns thunks (see section III.1), thus it is not a metafunction operating on template metaprogramming values. It is a template internally used by let. The result of this is wrapped with syntax<...> to turn the result of strict_let into a template metaprogramming value. This pattern – unwrapping the values, doing the computation and wrapping the result – is similar to the way operations on wrapped scalars work. let_in_syntax is the template that is doing the substitution itself.

```
template <class A, class E1, class E2>
struct let_in_syntax : let_impl<A, E1, E2> {};

template <class A, class E1>
struct let_in_syntax<A, E1, A> { typedef E1 type; };
```

let_in_syntax passes its arguments to let_impl in almost all cases, except for that when the expression to do the substitution in is the variable to substitute. In this case the result is the expression to replace the variable with. This case is implemented by the specialisation of let_in_syntax. It is important not to inherit from E1, but to return it as the result. This guarantees, that evaluating let_in_syntax will not accidentally evaluate E1. These are things metafunctions operating on template metaprogramming values don't need to worry about, but let_in_syntax belongs to the few metafunctions that can not operate on template metaprogramming values.

let_in_syntax implements the substitution of the variables. When the expression to do the substitution in is not a variable, it should be left unchanged. This is implemented by let_impl:

```
template <class A, class E1, class E2>
struct let_impl { typedef E2 type; };
```

This returns E2 and makes sure that it is only returned but not evaluated. This solution deals with the cases when the expression to do the substitution in is a simple one, such as var<x_> or mpl::int_<13>. When it is a complex one and consists of metafunction calls, such as mpl::plus<var<x_>, mpl::int_<13>>, let_in_syntax should recurse into it until only simple expressions remain, that are easy to deal with. This recursion is implemented using partial template specialisation and template template arguments. A complex expression looks like the following:

```
F<T1, T2, T3, ..., Tn>
```

F is a template metafunction – a template class – taking n arguments, T1 ... Tn are angle bracket expressions. let_impl is specialised to handle these cases:

```
template <
  class A,
  class E1,
  template <class, ...., class> class F,
  class T1, ..., class Tn
>
struct let_impl<A, E1, F<T1, ..., Tn>> {
  typedef
    F<
      typename let_in_syntax<A, E1, T1>::type,
      // ...
      typename let_in_syntax<A, E1, Tn>::type
    > type;
};
```

This code snippet uses a template class template argument, F to represent the metafunction being called and class template arguments to represent the angle bracket expressions being passed to it. It does the substitution of the arguments by calling let_in_syntax recursively and produces a new angle bracket expression calling the same metafunction, F with the substituted arguments.

n is a fixed number in the above example, which means that it implements recursion for calls of metafunctions with arity n. For every arity a new specialisation has to be made. The Boost.Preprocessor library [34] provides tools to automatically generate these specialisations.

Using two template classes – let_in_syntax and let_impl – may seem to be unnecessary for the first time, but it is important to avoid ambiguity when let is used. Adding all specialisations to let_in_syntax and not using let_impl would make the following instantiation ambiguous:

```
let_in_syntax<var<x_>, int_<13>, var<x_>>
```

This could match any of the following specialisations:

```
template <class A, class E1>
struct let_in_syntax<A, E1, A>;

template <class A, class E1, template<class> class F, class T1>
struct let_impl<A, E1, F<T1>>;
```

The compiler would not be able to decide if it should recurse into `var<x_>` or replace it with `E1`. Having two layers – `let_in_syntax` and `let_impl` – helps the compiler. When it has to choose a specialisation of `let_in_syntax` it may do the substitution or choose the general case. When the general case is selected, it triggers the instantiation of `let_impl`, which has specialisations implementing the recursion into the expression.

Boxed values

Boxed values wrap classes that are not prepared for template metaprograms, thus metaprograms should not look into boxed values. For let expressions it means that `let` should not recurse into boxed values. To ensure this, `let_impl` of the above implementation should be specialised:

```
template <class A, class E1, class V>
struct let_impl<A, E1, box<V>> : box<V> {};
```

This specialisation leaves boxed values unchanged. Thus for example

```
let<x, syntax<mpl::int_<13>>, box<x>>::type
```

returns `box<x>` instead of `box<mpl::int_<13>>`, since `box` protects its content from being substituted by a let expression. `syntax` is a special way of boxing, thus, `let` should also leave syntaxes inside syntaxes unchanged. This is implemented by another specialisation of `let_impl`:

```
template <class A, class E1, class E2>
struct let_impl<A, E1, syntax<E2>> : syntax<E2> {};
```

This specialisation matches cases when the expression to do the substitution in is a nested `syntax<...>` – the outer `syntax` wrapper has already been removed when `let_impl` is instantiated. It leaves the inner `syntax` unchanged. Having this specialisation, the following expression

```
let<
  x, syntax<mpl::int_<13>>,
  syntax<mpl::plus<x, syntax<x>>>
>::type
```

returns `syntax<mpl::plus<mpl::int_<13>, syntax<x>>>`, since the first occurrence of `x` is substituted, but the second one is inside a nested `syntax` and is left unchanged. The special handling of boxed values introduces a difference between `let` and `let_c`. For example:

```
eval_let<a, syntax<mpl::int_<13>>, syntax<
  let<b, syntax<mpl::int_<11>>, syntax<mpl::plus<a, b>>>
>>
```

The above example contains two nested let expressions. Since the body of the inner let expression, `syntax<mpl::plus<a, b>>` is wrapped by `syntax`, it is treated by `eval_let` as a boxed value and is not processed, thus evaluating the above expression would try evaluating `mpl::plus<a, mpl::int_<11>>`. However, by using `let_c`, one does not need to wrap the body with `syntax`:

```
eval_let_c<a, mpl::int_<13>,
  let_c<b, mpl::int_<11>, mpl::plus<a, b>>
>
```

The above example is the same as the previous one, but it uses `eval_let_c` and `let_c` instead of `eval_let` and `let`. As a result of this, the body of the inner let expression is not wrapped with `syntax` and the reference to `a` inside it gets substituted by the outer let expression.

This section has presented an implementation of preparing `let` for handling two types of boxed values. When further solutions for boxing values are developed in the future, `let` will need to be prepared for them. However, preparing `let` means adding a new specialisation to a template class, thus these updates can be done without changing already written code.

Variable hiding

One way of looking at let expressions is that they define a variable in the scope of a syntax. Nested usage of `let` may result in *variable hiding*. For example:

```
let<
  x, syntax<mpl::int_<11>>,
  syntax<
    mpl::plus<x, let<x, syntax<mpl::int_<13>, syntax<x>>>>
  >>
```

This expression substitutes the variable `x` with `mpl::int_<11>` inside the expression `mpl::plus<x, let<x, syntax<mpl::int_<13>, syntax<x>>>>`. A sub-expression of it, `let<x, syntax<mpl::int_<13>>, syntax<x>>` is also a let expression substituting the same variable. This inner let expression hides the variable `x` – inside it, the value `mpl::int_<13>` instead of the value `mpl::int_<11>` is bound to it.

The implementation of `let` presented in this dissertation would leave the body of the inner let expressions unchanged, as it is protected by an inner `syntax`, but the `x` argument of the inner `let` would be substituted producing the following result:

```
syntax<
  mpl::plus<
    mpl::int_<11>,
    let<mpl::int_<11>, syntax<mpl::int_<13>>, syntax<x>>
  >
>
```

This is an invalid let expression, since it uses `int_<11>` as a variable to bind to. `let` is prepared for variable hiding by adding a new specialisation to `let_impl`:

```
template <class A, class E1a, class E1b, class E2>
struct let_impl<A, E1a, let<A, E1b, E2>> : {
  typedef let<A, E1b, E2> type;
};
```

This specialisation handles the case when the variable used by the inner and the outer let expressions are the same, in which case it leaves the let expression unchanged. Given that a number of helper templates, such as `let_c`, `eval_let`, etc have been introduced, `let_impl` needs to be specialised for them as well.

III.5.4 Lambda expressions

The more generic functions are used in development, the more small utility functors implementing custom logic for the generic algorithms are needed. For example the algorithm `std::transform` and its metaprogramming equivalent provided by Boost.MPL, `mpl::transform` change each element of a sequence. They take a sequence and a functor changing one element of the sequence as arguments and produce a new sequence by applying the functor on all elements of the original one. `transform` and various other similar functions are useful in many places, but the small utility functions implementing the custom bits need to be provided, such as the transformation of one element of a sequence. These small functions contain bits of the business logic of the application. By implementing them as utility functions some parts of the business logic are moved to different locations of the source code, making it more difficult to follow the logic of a program when one has to read it later.

Lambda functions provide a solution for this issue. It is a technique for implementing utility functions in-place in the middle of an expression. These functions have no names unless they are stored in variables. This solution makes implementing the functions for generic algorithms in-place possible.

This section presents a solution for implementing lambda expressions that can be created, passed around and evaluated at compile-time. They are similar to lambda expressions introduced in C++11 [30], but they represent compile-time computations.

As developers use generic functions in complex situations, they need to construct complex functors as lambda expressions. Complex lambda expressions may contain nested lambda expressions. Let's consider the following data structure:

```
typedef
  mpl::list<
    mpl::list_c<int, 1, 2>,
    mpl::list_c<int, 3>
  >
  list_in_list;
```

Let's use `mpl::transform` to double every element of the nested lists. The Boost.MPL library provides a lambda implementation, `mpl::lambda`. The following example uses it to construct the functor for `mpl::transform`:

```
mpl::transform<
  list_in_list,
  mpl::lambda<
    mpl::transform<
      mpl::_1,
      mpl::lambda<
        mpl::times<mpl::_1, mpl::int_<2>>
      >::type
    >
  >::type
>::type
```

`mpl::_1` is used in the outer and in the inner lambda expression as well. Implementing it is possible but it makes understanding the code more difficult, since the occurrences of `mpl::_1` refer to different things.

Let's consider another example: using `mpl::transform`, every number should be increased by the length of the list containing the number. The expected result is

```
typedef
  mpl::list<
    mpl::list_c<int, 3, 4>, // length is 2
    mpl::list_c<int, 4> // length is 1
  >
  result_of_second_example;
```

In this case, the argument of the outer lambda expression has to be used in the inner one. Using the lambda expressions provided by Boost.MPL, workarounds are needed to implement it. The inner lambda expression has to take two arguments: the value of the outer expression's argument and the real argument of the inner expression. Some currying solution, such as the one presented in section III.2 has to be used to hide the first argument and make it work with the generic algorithm, `mpl::transform`. Given the complexity of this solution, developers are likely to create small helper functions instead. This approach suffers from the issue of having the business logic at different locations of the source code.

One may look at a lambda expression as a syntax with a list of arguments, which is a list of variables. This dissertation implements this using the `lambda` and `lambda_c` template classes. They can be used to implement the lambda expression for the above transformation:

```
mpl::transform<
  list_in_list,
  lambda_c<
    i,
    mpl::transform<i, lambda_c<j, mpl::plus<mpl::size<i>, j>>>
  >
>::type
```

`lambda_c` takes the list of variables representing the formal arguments of the lambda function as its arguments. The last argument of `lambda_c` is an angle-bracket expression, which is the body of the lambda expression using the formal variables. Since the variable names are chosen by the developer, it is easy to refer to variables used by outer lambda expressions. `lambda_c` is a helper template class wrapping `lambda` operating on syntaxes. The connection between them is the same as the connection between `let` and `let_c`.

The implementation of `lambda`

For an arity `n`, a template class `lambdan` is implemented. Having those template classes, `lambda` is implemented the following way:

```
struct no_argument;

template <class T1 = no_argument, ..., class Tk = no_argument>
struct lambda;
```

A special class, `no_argument` is introduced representing that no value has been specified for a template argument. All arguments of `lambda` has this as the default value, thus `lambda<j, syntax<mpl::plus<i, j>>>` refers to `lambda<j, syntax<mpl::plus<i, j>>, no_argument,...,no_argument>`. `lambda` is specialised for every arity:

```
template <class T1, ..., class Tn, class Body>
struct lambda<T1, ..., Tn, Body, no_argument,...,no_argument> :
  lambdan<T1, ..., Tn, Body> {};
```

These specialisations assume that `n` variables and the body of the lambda expression are provided and the rest of the arguments are `no_argument` classes. They instantiate the `lambdan` template implementing a metafunction class with arity `n`.

The maximum arity of lambda expressions is limited by the choice of `k`, the number of arguments of `lambda`. The Boost.Preprocessor library [34] provides tools to automatically generate `lambda` itself and the specialisations. Using them also makes it possible to turn this upper limit into an argument of the compilation process, thus it can be increased if needed. Using *variadic templates* offered by C++11, the same thing could be implemented in a more flexible way but projects not using C++11 would not be able to use `lambda`.

`lambda` for a fixed arity is implemented using the tools presented so far. It is a template metafunction class taking `n` arguments, where `n` is the arity of the lambda expression.

```
template <class T1, ..., class Tn, class Body>
lambdan : tmp_value<lambdan<T1, ..., Tn, Body>> {
  template <class A1, ..., class An>
  struct apply :
    eval_syntax<
      let<T1, A1,
        let<T2, A2,
          // ...
          let<Tn, An, Body>
          // ...
        >>> {};
};
```

This solution defines a metafunction class. It uses `tmp_value` to make the result a template metaprogramming value. The `apply` template metafunction turns it into a metafunction class. It uses `let n` times to replace the variables in the body of the lambda expression with the parameter values the metafunction class was called with. Once all the variables have been substituted, `eval_syntax` is used to evaluate the body of the lambda expression and provide the result.

Variable hiding

When a lambda expression is used inside a let expression, the lambda expression may use the same variables as the let expression. For example:

```
let<a, syntax<mpl::int_<13>>, syntax<lambda<a, syntax<a>>>>
```

The body of the above let expression contains a lambda expression and both of them use the variable `a`. `let` does not substitute the variable `a` in the body of the lambda expression, as it is protected by an inner `syntax` block, but it does the substitution for the argument of the lambda expression, which gives the following result:

```
syntax<lambda<mpl::int_<13>, syntax<a>>>
```

The above code snippet shows what `let` builds from the previous example. The argument of the lambda expression is `a`, thus it gets replaced with `mpl::int_<13>`, however, the body of the lambda expression is inside `syntax`, thus it remains unchanged.

As presented in section III.5.3, `let` can be prepared for handling boxing classes by specialising the template class `let_impl`. It can also be prepared for `lambda` by adding further specialisations. For example:

```
template <class A, class E1, class A1, class A2, class F>
struct let_impl<A, E1,
  lambda<A1, A2, F, no_argument, ..., no_argument>
> :
  lambda<A1, A2,
    mpl::if_<
      mpl::contains<mpl::vector<A1, A2>, A>,
      F,
      let_in_syntax<A, E1, F>
    >,
    no_argument, ..., no_argument
  >
{};
```

The above code snippet shows how to prepare `let` for lambda expressions expecting two arguments. The arguments of the lambda expression, `A1` and `A2` are left unchanged. If any of them is the same as the one being substituted by the lambda expression, the body is left unchanged. Otherwise it is substituted using `let_in_syntax`. Substituting the body using `let_in_syntax` instead of `let` ensures, that when the body is a `syntax` and not an expression evaluating to one, the recursion stops. The code checks if any of the formal arguments of the lambda is `A` by constructing a vector from the arguments and checking if it contains `A`.

The above specialisation prepares `let` for lambda expressions expecting two arguments. A different specialisation has to be written for every arity. Writing these specialisations manually is error prone, but the Boost.Preprocessor [34] library provides tools supporting the automatic generation of the specialisations.

The approach presented in this section implements variable hiding when lambda expressions are used inside let expressions. Using a let expression inside a lambda expression also causes variable hiding in some cases, just like using lambda expressions inside lambda expressions. However, since the variable substitution of lambda expressions has been implemented using `let`, which is already prepared for nested `let`s and `lambda`s, it works without further changes.

Currying

As lambda expressions are higher order template metafunctions, they should support currying as well. This section presents how to add currying support to `lambda`. It makes every lambda expression implemented using `lambda` support currying. For example:

```
typedef lambda_c<a, b, mpl::plus<a, b>> add;
typedef add::apply<mpl::int_<1>> inc;

typedef inc::apply<mpl::int_<12>>::type int13;
```

The above code snippet shows how `lambda_c` with currying support works. It defines the `add` metafunction class using `lambda_c`. `add` adds two values by calling `mpl::plus`. However, `add::apply<mpl::int_<1>>` calls this code with one argument only which returns another metafunction class taking one argument and adding `mpl::int_<1>` to it. This new metafunction class increases its argument by 1. It is given a name: `inc`. The example shows how to use `inc` by calling it with the argument `mpl::int_<12>` to get `mpl::int_<13>`.

Currying support for `lambda` is implemented by keeping the list of parameters in a container – eg. `mpl::list` – and when the metafunction class is called, doing the substitution of the available parameters and removing them from the container. Once all of the parameters have been substituted, the body is evaluated and the result is returned. The implementation is not presented here in detail. A reference implementation can be found in [59].

III.5.5 Recursive let expressions

Let's consider the following example which bounds a higher order function implementing factorial to the name `fact` in the expression `mpl::apply<fact, mpl::int_<3>>`:

```
struct fact_;
typedef var<fact_> fact;

let_c< // attempt to implement fact
  fact,
  lambda_c<
    n,
    mpl::eval_if<
      mpl::equal_to<n, mpl::int_<0>>,
      mpl::int_<1>,
      mpl::times<
        mpl::apply<
          fact,
          mpl::minus<n, mpl::int_<1>>
        >,
        n
      >
    >
  >,
  mpl::apply<fact, mpl::int_<3>>
>
```

The above code snippet does not work because the factorial function is implemented recursively but `let_` expects the expression to bind not to rely on recursion. `let_c` does not bind the name `fact` in factorial's implementation to the name `fact`, thus recursive calls will not work.

Let expressions in Haskell support recursion. There are languages [70, 14] which provide a different syntax for recursive (typically called `letrec`) and non-recursive let expressions. This section follows this approach in C++ template metaprogramming as well and the recursive version is implemented separately, using the non-recursive version in its implementation. The recursive version of `let` will be called `letrec`. The declaration of `letrec` is the following:

```
template <class A, class E1, class E2>
struct letrec;
```

`E1` needs to be bound to `A` in the scope of `E1` and the resulting expression needs to be bound to `A` in the scope of `E2`. The following code does this:

```
// attempt to implement letrec
template <class A, class E1, class E2>
struct letrec :
  let<A, let<A, E1, E1>, E2>
{};
```

The problem with this implementation is that the expression `E1` bound to `A` in the scope of `E1` – in the inner `let` binding – may use `A` as well. Thus, `E1` has to be bound to `A` recursively in the scope of `E1`.

```
template <class A, class E1, class E2>
struct letrec :
  let<A, letrec<A, E1, E1>, E2>
{};
```

The above code snippet uses `letrec` recursively to implement recursive binding of the expression `E1` to the name `A` in the expression `E1` itself. Since binding happens lazily, recursion happens only when `E1` uses the name `A`. When `E1` does not use the name `A`, `letrec` is not evaluated again and the recursion stops. The result of this recursive binding is an expression. This expression is bound to the name `A` in the scope of `E2`. Since `E2` is not bound to any name, no recursive binding is needed at this step, thus `let` can be used. The following example shows how to use `letrec` to implement the factorial example:

```
letrec< // attempt to implement fact
  fact,
  lambda_c<n,
    mpl::eval_if<
      mpl::equal_to<n, mpl::int_<0>>,
      mpl::int_<1>,
      mpl::times<
        mpl::apply<
          fact,
          mpl::minus<n, mpl::int_<1>>
        >, n>>>,
  mpl::apply<fact, mpl::int_<3>>
>
```

`letrec` binds the recursive implementation of the factorial function to the name `fact` in the scope of the function as well, thus it works. However, it relies heavily on lazy evaluation, thus it is important that all functions used in the body of the factorial function evaluates its arguments lazily. The functions Boost.MPL provides expect eagerly evaluated arguments [57, 58], thus they need to be wrapped with code that lazily evaluates the arguments and then calls the functions of the Boost library. The following example shows how to use the `lazy` template presented in section III.1.3 here:

```
letrec<
  fact,
  lambda_c<n,
    lazy<
      mpl::eval_if<
        equal_to<lazy_protect_args<n>, mpl::int_<0>>,
        mpl::int_<1>,
        lazy_argument<
          mpl::times<
            mpl::apply<
              fact,
              mpl::minus<lazy_protect_args<n>, mpl::int_<1>>
            >,
            lazy_protect_args<n>>>>>>,
  apply<fact, mpl::int_<3>>
>
```

This is finally a working version of the factorial example, that uses only lazy metafunctions.

III.6 Pattern matching

Trying to evaluate the template metaprogramming expression `mpl::divides< mpl::int_<1>, mpl::int_<0>>::type` emits a compilation error due to the division by zero and this error is not recoverable. Since the divider may not be known at compile-time, programs need to be prepared for handling this. One way of doing it is checking the divider before calling `divides` (this is what Boost.MPL expects). Another approach is preparing `divides` to handle these situations and report the problem to the caller. For this second approach, Haskell provides `Maybe`, presented in section III.3. That section has presented the following approach for implementing `Maybe` in C++ template metaprogramming:

```
template <class A> struct just
  { typedef just<typename A::type> type; };

struct nothing { typedef nothing type; };
```

It represents a value that may contain a result or represent failure. Here is a safe version of `divides` using `Maybe`:

```
template <class A, class B>
struct safe_divides :
  mpl::eval_if<
    typename mpl::equal_to<B, mpl::int_<0>>::type,
    nothing,
    just<divides<A, B>>
  > {};
```

This code snippet checks if `B` is zero. When it is, it returns `nothing`, otherwise it does the division, wraps the result with `just` and returns that. It makes use of the fact, that constructors of algebraic data types behave as lazy template metafunctions: instantiating `just<divides<A, B>>::type` instantiates `just<divides<A, B>::type>` based on the above definition of `just`.

Users of `safe_divides` need a way of checking if the result is `nothing` and if not, getting the value wrapped by `just`. This can be demonstrated using an example function, `divide_if_possible`. This function takes two arguments, `A` and `B` and returns `A / B` when the division is valid. This function always returns numbers, even when the division is invalid. This example returns `A` in that case. When it is implemented using `safe_divides`, it has to check the result and unwrap it.

A commonly used [25, 37] approach for implementing the verification and unwrapping in template metaprogramming is partial template specialisation.

```
template <class A, class ResultOfSafeDivides>
struct divide_if_possible_impl;

template <class A>
struct divide_if_possible_impl<A, nothing> : A {};

template <class A, class R>
struct divide_if_possible_impl<A, just<R>> : R {};

template <class A, class B>
struct divide_if_possible :
  divide_if_possible_impl<A, typename safe_divides<A, B>::type>
{};
```

The above code snippet defines `divide_if_possible_impl`, a helper template metafunction taking the result of `safe_divides` as one of its arguments. This helper metafunction is specialised for `nothing` and `just<R>` and the specialisations handle the different cases. The helper metafunction implements something similar to the `case` structure of Haskell.

The drawback of this approach is that the developer has to write a helper metafunction for every pattern matching. It introduces a large amount of helper metafunctions, which pollute the namespace(s) and make the code more difficult to read and maintain.

III.6.1 Using syntaxes for pattern matching

This section presents how to use syntaxes with variables to implement pattern matching. The angle bracket expressions wrapped by `syntax` represent the patterns, the variables represent the parts that need to be unwrapped. For example the syntax `just<x>` represents a pattern for a value wrapped by `just`. The variable `x` represents the value to be unwrapped. The syntax `syntax<nothing>` represents the `nothing` pattern.

A template metaprogramming value, such as `just<int_<13>>` can be matched against a pattern (which is a syntax). This matching is implemented as a template metafunction, such as `match` taking the pattern and the template metaprogramming value as arguments. When the matching succeeds, it returns an `mpl::map` mapping variables to values. When the matching fails, it returns a special value, `nothing` indicating this.

A helper metafunction is needed for the implementation, that can decide if the result of `match` is a map or a failure. This is implemented using template specialisation.

```
template <class MatchResult>
struct matched : mpl::true_ {};

template <>
struct matched<nothing> : mpl::false_ {};
```

This metafunction returns `mpl::false_` for the value indicating failure and `mpl::true_` for other values. `match` is implemented the following way:

```
template <class Pattern, class Expression>
struct strict_match;

template <class Pattern, class Expression>
struct match :
  strict_match<
    typename Pattern::type,
    typename Expression::type
  > {};
```

This implementation starts with evaluating both the pattern and the expression to make `match` a lazy metafunction. The evaluated results are passed to `strict_match`, which does the pattern matching.

```
template <class Pattern, class Expression>
struct strict_match<syntax<Pattern>, Expression> :
  match_impl<Pattern, Expression> {};
```

`strict_match` unwraps the pattern and passes it together with the expression to `match_impl`.

```
template <class Pattern, class Expression>
struct match_impl : nothing {};

template <class Expression>
struct match_impl<Expression, Expression> : mpl::map<> {};
```

`match_impl` returns `nothing` when the pattern and the expression are different types and an empty map when they are the same. For example the value `mpl::int_<13>` matches the unwrapped pattern `mpl::int_<13>`, but not the unwrapped pattern `mpl::int_<11>`, or `nothing`. To support variables, a further specialisation needs to be added:

```
template <class Var, class Expression>
struct match_impl<var<Var>, Expression> :
  mpl::map<mpl::pair<var<Var>, Expression>> {};
```

When the pattern is a variable, the result of the pattern matching is a one element map, mapping the variable to the expression itself. For example matching the value `mpl::int_<13>` against the unwrapped pattern `var<x_>` gives a map mapping `var<x_>` to `mpl::int_<13>`.

In most cases, like matching against the pattern `just<var<x<_>>>`, the variables are arguments of a constructor used in the pattern. To handle these cases as well, `match_impl` has to recurse into the expression and the value at the same time. This is implemented using partial template specialisation and template class template arguments.

```
template <
  template <class, class> class T,
  class P1, class P2,
  class E1, class E2
>
struct match_impl<T<P1, P2>, T<E1, E2>> :
  merge_maps<match_impl<P1, E1>, match_impl<P2, E2>> {};
```

The above code snippet shows how recursion is implemented for constructors taking two arguments. `match_impl` is specialised for template classes taking two arguments. The same template class – the same constructor – has to be used in the pattern and by the value, but their arguments may differ. The arguments of the constructor are called `P1` and `P2` in the pattern and `E1`, `E2` in the value. The arguments used in the pattern and the value are matched against each other. The results of these matchings need to be merged. This merging is implemented by `merge_maps`. What it has to be careful with is:

- When any of the sub-matchings fail, the entire pattern matching fails. For example the value `cons<mpl::int_<13>, nil>` can not be matched against the pattern `cons<mpl::int_<11>, nil>`.
- When two sub-matchings try to map different values to the same variable, the entire matching fails, as all occurrences of a variable need to match the same value in the entire expression. For example the value `cons<mpl::int_<13>, nil>` can not be matched against the pattern `cons<var<x_>, var<x_>>`.

The implementation of `merge_maps` is not presented here. A reference implementation of `match` can be found in [59].

III.6.2 Let expressions

A pattern can contain multiple variables and when a pattern matching succeeds, these variables are all bound to a value. For example matching the value `mpl::pair<mpl::int_<11>, mpl::int_<13>>` against the pattern `syntax<mpl::pair<var<x_>, var<y_>>>` binds `mpl::int_<11>` to `var<x_>` and `mpl::int_<13>` to `var<y_>`. Some languages, such as Haskell allow pattern matching in let expressions, thus in Haskell one may write

```
-- defining some helper type
data PairT a b = Pair a b

-- doing the pattern matching in a let expression
let
  Pair x y = Pair 11 13
in
  x + y
```

This code snippet binds the value `11` to `x` and `13` to `y` in the scope of the expression `x + y`. This section presents how to provide the same functionality for template metaprogramming by combining let expressions and pattern matching. A metafunction `match_let` is provided:

```
template <class Pattern, class Value, class Body>
struct match_let;
```

It takes three arguments: the pattern to use, the value to match against the pattern and the syntax to bind the variables in the scope of. It does the following:

- Matches the value against the pattern. This results in a number of variable bindings.
- Substitutes all occurrences of the variables with the values they are bound to in the body.

The result of it is the substituted body. It is implemented by combining `let` and `match`, the implementation details are not presented here. A reference implementation can be found at [59]. By using `match` and `match_let`, it is possible to implement metafunctions like `divide_if_possible` without writing helper metafunctions:

```
template <class A, class B>
struct divide_if_possible :
  mpl::eval_if<
    typename matched<
      match<
        syntax<nothing>,
        safe_divides<A, B>
      >
    >::type,

    A,
    match_let<
      syntax<just<r>>, safe_divides<A, B>,
      syntax<r>
    >
  >
{};
```

To check if it is possible to do the division, this approach matches the result of `safe_divides` against the pattern `syntax<nothing>`. When it is not possible, `divide_if_possible` returns the first argument, `A`. Otherwise it uses `match_let` to unwrap the value wrapped with `just` and returns it.

III.6.3 Case expressions

Even though the above implementation of `divide_if_possible` does not require helper metafunctions, it is still difficult to understand the logic of the code by looking at it. Many functional languages, to simplify code written in them provide *case expressions*. These expressions have a value, such as the result of `safe_divides` and a number of cases, each of them consisting of a pattern and a body. The value is matched against the patterns in order and the body of the first one that matches is evaluated and returned. For example

```
divide_if_possible a b =
  case safe_divides a b of
    Nothing -> a
    Just r -> r
```

This implementation of `divide_if_possible` in Haskell uses a case expression to process the result of `safe_divides`:

- When the division is not possible and the result is `Nothing`, it returns the first argument, `a`.
- When the division is possible, the result is wrapped with `Just`. However, in the pattern of the case expression it is allowed to use variables which can be used in the body of that case. Using this, the code unwraps the value returned by `safe_divides` and returns the result of the division.

This section presents how to implement a similar construct in template metaprogramming using syntaxes, pattern matching and `match_let`.

```
struct no_case;

template <class Value,
          class Case1=no_case, ..., class CaseN=no_case>
struct case_;
```

This is a metafunction taking a value and a number of cases. Instances of the following template class describe these cases:

```
template <class Pattern, class Body> struct matches;
```

This template class has two arguments: the pattern to match against the `Value` argument of `case_` and the body to evaluate when the pattern matches. Using the above implementation a case expression can have `N` cases, however the case arguments have a default value, `no_case` which means that that case is not provided. This makes it possible to have less cases than `N`. The Boost.Preprocessor library [34] provides tools to automatically generate the code of `case_` and to turn `N` into a compilation argument, thus one can increase it when it is needed. The above approach is not using features like variadic templates [30, 72] provided by C++11. Using them could make the code more flexible, but it would not work with earlier compilers not supporting these features. The implementation of `case_` has to do the following:

- Try matching the value against the patterns of the cases provided.
- When one matches, in the corresponding body all occurrences of the pattern's variables need to be substituted with the values the pattern matching assigned to them.

The details of the implementation are not presented here. A reference implementation can be found in [59]. Using `case_` the implementation of the above `divide_if_possible` example becomes easier to read:

```
template <class A, class B>
struct divide_if_possible :
  eval_syntax<
    case_< safe_divides<A, B>,
      matches<syntax<nothing>, syntax<A>>,
      matches<syntax<just<r>>, syntax<r>>
    >
  >
{};
```

This code snippet checks the result of `safe_divides` using `case_`. When it is `nothing`, it returns the first argument, `A`. When it is a value wrapped with `just`, it returns the value. As `case_` returns a syntax – the selected body in which the variables of the corresponding pattern have been substituted – it has to be evaluated using `eval_syntax`. A helper metafunction, `eval_case` is also provided to make the above code simpler. It is the combination of `eval_syntax` and `case_` and can be used the following way:

```
template <class A, class B>
struct divide_if_possible :
  eval_case< safe_divides<A, B>,
    matches<syntax<nothing>, syntax<A>>,
    matches<syntax<just<r>>, syntax<r>>
  >
{};
```

This code snippet does the same as the previous one, but uses `eval_case` instead of `case_` inside `eval_syntax`. Similarly to `let` and `let_c`, `matches_c` can also be provided to offer an easier way of implementing cases. The details of it are not presented here.

This section has presented how to provide pattern matching, the approach used by many functional programming languages for finding out which constructor a value was created with and accessing the constructor arguments in template metaprogramming, an environment not explicitly prepared for handling this. Using it makes the implementation of metaprograms simpler.

III.7 Summary

This chapter has presented how to provide a number of commonly used approaches of functional languages in C++ template metaprogramming and how they help the development and maintenance of template metaprograms.
Thesis 1: I have evaluated the connection between C++ template metaprogramming and functional programming languages. Based on the results I have developed methods for supporting template metaprogrammers using the functional paradigm explicitly. (chapter III)

Thesis 1.1: I have shown the importance of laziness in template metaprogramming and developed an automated adaption method to use non-lazy metafunctions in a lazy way. (section III.1)

Thesis 1.2: I developed a method for effective implementation of currying in C++ template metaprogramming. (section III.2)

Thesis 1.3: I have developed a method for representing Haskell-like algebraic data-types in C++ template metaprogramming. (section III.3)

Thesis 1.4: I have developed a method for representing Haskell typeclasses in C++ template metaprogramming. (section III.4)

Thesis 1.5: I have developed a method to handle template metaprogramming expressions as first class citizens, ie. they can be stored, passed as parameters or returned by functions. This method enables the implementation of let expressions and provides a more convenient way of implementing lambda expressions than what Boost.MPL's lambda expression implementation, a widely used solution offers. (section III.5)

Thesis 1.6: I have implemented an alternative method for pattern matching in C++ template metaprogramming. This enables the implementation of case expressions. (section III.6)

Table III.1: Related publications

	[53]	[57]	[58]	[60]	[61]	[63]	[64]	[67]
1.1		×	×		×	×		
1.2		×	×			×		
1.3	×				×		×	×
1.4							×	×
1.5				×	×			
1.6					×			

Chapter IV

Monads

In functional programming *monad*s [48, 39, 38] are a tool for abstracting computation. Users of a monad combine different functions together using the operations provided by the monad. The code combining these functions together is generic, it is implemented without knowing what these functions will be.

A monad decorates functions that are evaluated in sequence to implement some general logic that is orthogonal to the functions themselves. An example of such orthogonal logic is error propagation in a sequence of functions – when one of them fails and returns an error, the error should be returned to the caller without evaluating the rest of the functions. Among error propagation, monads have various use cases. A few examples:

- Input and output is implemented using a special monad, called the *IO monad* [48] in Haskell.
- Monads help the implementation of parser combinators. Parsers can be combined in a monadic way and the monadic framework takes care of a large amount of boilerplate code during this process.
- Monads simplify the implementation of pure code doing logging or producing other type of output. A monad takes care of collecting that output.
- Monads simplify error propagation in complex code. By using them, the error propagation logic can be separated from the business logic. They can be used to simulate exceptions in pure code.
- Monads simplify the implementation of pure code operating on a state. Different types of monads handle mutable and immutable states.

Following the object oriented programming paradigm [9] one can abstract either the data a computation is working on or the computation that works on a piece of data.

- To abstract the data a code snippet is working on one can use the *Composite* design pattern [17]. When it is used, a computation doesn't need to know the real type of the object it is working on.

- To abstract the computation, one can use the *Command* design pattern [17]. It abstracts parts of a larger computation away. The rest of the code is implemented in a way that is not aware of what these parts are doing.

A monad operates on functions that are passed around as values taking advantage of the fact that functions are first class citizens in functional programming. The Command design pattern uses inheritance and runtime polymorphism to be able to pass small code snippets around as values [48].

In Haskell a monad is implemented by a typeclass, called `Monad`. It takes a type constructor as argument. The type constructor has to take one argument to produce a type. Instances of `Monad` are called *monadic types*, values of those types are called *monadic values*. The typeclass requires the following operations to be implemented:

```
class Monad m where
  return :: a -> m a
  fail :: String -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b

  a >> b = a >>= \_ -> b
  fail = error
```

Two operators are required, both of them taking two arguments:

- `>>=`, taking a monadic value and a function mapping a value of some type to a monadic value. The operator builds a new monadic value. This operator may decorate the function or decide not to evaluate it at all. Since the first argument of this operator can be the result of another call of this operator, it can be used to call a number of functions mapping values to monadic values in a sequence. This operator implements some general logic that is orthogonal to what the functions that are evaluated in a sequence do.

- `>>`, taking two monadic values and building a new one. The typeclass provides a default implementation for this function that calls the `>>=` operator with a function that always returns the second argument of `>>`.

As an addition, two functions are required:

- `return`, taking a value of some type and returning a monadic value. The purpose of this function is to lift a value into the monad.

- `fail`, taking a string and returning a monadic value. The returned value has to break a sequence of functions built with the `>>=` operator. This function has a default implementation that calls `error`, which generates an exception [48]

Monads add value to template metaprogramming when a syntactic sugar, the *do notation* is ported from Haskell to C++ template metaprogramming. This chapter presents an approach for simplifying template metaprograms based on monads.

IV.1 Implementation of monads

In C++ template metaprogramming monadic values are represented as a *set of template metaprogramming values*. Since template metaprogramming is weakly typed, this set in defined in an informal way – in a comment or in the documentation. In Haskell the compiler verifies if a value is monadic or not, while the C++ compiler can not do it in template metaprogramming. On the other hand, this makes the definition of monadic values more flexible – there are sets of monadic values that can be defined in template metaprogramming but not in Haskell. Monads are implemented using typeclasses requiring the following metafunctions:

- `return_`, taking some metaprogramming value as argument and returning a monadic value. This is the equivalent of `return`.

- `bind`, taking a monadic value and a metafunction class as arguments and returning a monadic value. The metafunction class takes some value and returns a monadic value. This is the equivalent of the `>>=` operator.

- `bind_`, taking two monadic values as arguments and returning a new monadic value. This metafunction evaluates its arguments lazily [57], thus it accepts nullary metafunctions returning monadic values as arguments. It can replace `bind` in cases where the metafunction class ignores its argument. This is the C++ template metaprogramming equivalent of the `>>` operator of Haskell.

- `fail`, taking some value as argument and returning a monadic value. Its purpose is reporting errors in monads. When it is used in a sequence of `bind` calls, the returned value should break the evaluation of the sequence. This function is called `fail` in Haskell as well.

Since C++ operator syntax can not be used in template metaprogramming, the operations are given names. A typeclass called `monad` is created as the equivalent of the `Monad` typeclass in Haskell:

```
template <class Tag>
struct monad;
  // Requires: return_::apply <T>, fail::apply <T>
  //           bind::apply <T, F>, bind_::apply <T, V>
```

As `>>` and `fail` have default implementations in Haskell, the template metaprogramming version provides a default implementation for `bind_` and `fail`:

```
template <class T> struct monadic_error {};

template <class Tag>
struct monad_defaults {
  struct bind_ : tmp_value<bind_> {
    template <class A, class B>
    struct apply :
      mpl::apply_wrap2<
        typename monad<Tag>::bind, A, mpl::always<B>
      > {};
  };

  struct fail : tmp_value<fail> {
    template <class T>
    struct apply : monadic_error<T>::failed {};
  };
};
```

`bind_`'s default implementation calls `bind` with a metafunction class that always returns `bind_`'s second argument. In Haskell `fail`'s default implementation uses `error`, which is something not available in C++ template metaprogramming. However, code that breaks the compilation process by accessing a non-existing nested type in a template class is used instead. The name of the nested class is likely to appear in the error message generated by the compiler, thus giving this class a meaningful name improves the quality of the error message. The example above uses a non-existing nested class called `failed`. Since it is trying to access a nested class in a template class instance, it doesn't generate any error message until it is instantiated. Every instance of `monad` has to publicly inherit from `monad_defaults` to get the default implementations.

Wrapper template metafunctions for the functions expected by the monad simplify the usage of this monad implementation. The tag of the monad has to be the first argument of these metafunctions.

```
template <class Tag, class T>
struct return_ :
  mpl::apply<typename monad<Tag>::return_, T>
{};
```

The above example shows how to implement a helper function for `return_`. The rest of the functions (`bind`, `bind_` and `fail`) can be implemented in a similar way.

Haskell has semantic expectations for monads [48] that are documented but cannot be verified by the compiler. The C++ template metaprogramming equivalents of these expectations are the following:

- *left identity*: `bind<Tag, return_<Tag, X>, F>` is equivalent to `mpl::apply<F, X>`.
- *right identity*: `bind<Tag,M,monad<Tag>::return_>` is equivalent to `M`.
- *associativity*: The expression `bind<Tag, M, lambda_c<x, bind<Tag, mpl::apply<F,x>,G>>>` is equivalent to `bind<Tag,bind<Tag,M,F>,G>`.

Similarly to Haskell, these expectations can not be verified automatically. It is the responsibility of the monad's author to satisfy these expectations.

IV.2 Monad variations

This section presents how to implement different types of monads available in Haskell in C++ template metaprogramming. The full implementation of these monads is part of Mpllibs [59].

IV.2.1 Maybe

Section III.3 presents how the `Maybe` type is implemented in Haskell:

```
data Maybe a = Just a | Nothing
```

It is implemented in C++ template metaprogramming the following way:

```
template <class A>
struct just : tmp_value<just<A>> {};

struct nothing : tmp_value<nothing> {};
```

The following implementation makes it an instance of the `monad` typeclass:

```
template <>
struct monad<maybe> : monad_defaults<maybe> {
  typedef lambda_c<t, just<t>> return_;
  typedef lambda_c<_, nothing> fail;

  typedef
    lambda_c<a, f,
      eval_case<a,
        matches_c<just<v>, mpl::apply<F, v>>,
        matches_c<nothing, a>
      >
    >
    bind;
};
```

`return_` wraps its argument with `just`, `bind` checks if its first argument, the result of the previous step in the sequence is `nothing`. When it is, it returns this value without calling the next step. Otherwise it unwraps the value from `just` and passes it to the next step in the chain. `fail` returns `nothing` to break the chain of `bind`s.

This monad implements some error propagation logic. It helps combining metafunctions using `Maybe` to report errors. The problem with this solution is that the monadic functions can not return any detail about the error.

IV.2.2 Either

The `Either` monad is a tool for error handling as well. In Haskell the following type is defined:

```
data Either a b = Left a | Right b
```

When it is used for error handling, `Left a` represents an error, `Right b` represents a result. Since `Left` has an argument as well, functions using `Either` for error reporting can report details describing what went wrong. `Either` is a monad instance as well, `bind` implements error propagation logic. The type constructors are implemented as template classes in C++ template metaprogramming:

```
template <class A> struct left;
template <class B> struct right;
```

These two constructors form an algebraic data-type, thus they are implemented based on section III.3. A new tag, `either` is needed for the `Either` monad. `either` is an instance of `monad`:

```
template <>
struct monad<either> : monad_defaults<either> {

  typedef lambda_c<t, right<t>> return_;

  typedef lambda_c<s, left<s>> fail;

  typedef
    lambda_c<a, f,
      eval_case<a,
        matches_c<right<v>, mpl::apply<f, v>>,
        matches_c<left<_>, a>
      >
    >
    bind;
};
```

`return_` wraps its argument with `right` to make it a result, `fail` wraps its argument with `left` to break the evaluation of the monad. `bind` propagates the error, when its first argument is `left`. When its first argument is `right`, it unwraps the value and calls the monadic function.

Using this monad for implementing error-handling makes it possible for a function to return information about what the problem was in case of errors.

IV.2.3 List

The list monad turns operations mapping elements to lists into operations transforming lists. Monadic values are lists of some type. `return_` creates a list with one element. `bind`'s first argument is a list. It calls the monadic function on all elements of this list and concatenates the resulting lists. Since the List monad doesn't deal with error handling, there is no reasonable way of overriding `fail`. It is implemented the following way:

```
typedef
  lambda_c<s, l,
    lazy<
      mpl::insert_range<
        already_lazy<s>,
        lazy_protect_args<mpl::end<s>>,
        already_lazy<l>
      >>>
  join_lists;

template <>
struct monad<list_tag> : monad_defaults<list_tag>  {
  typedef lambda_c<t, boost::mpl::list<t>> return_;

  typedef
    lambda_c<a, f,
      mpl::fold<
        mpl::transform_view<a, f>,
        mpl::list<>,
        join_lists
      >>
    bind;
};
```

The `return_` operation builds a one element list from its argument. The `bind` operation applies its second argument, the function on all elements of the list, which is its first argument. This application is implemented using the `mpl::transform` metafunction provided by Boost.MPL. The result of this is a list of lists, which needs to be concatenated. This happens by folding over this list using a helper metafunction class, `join_lists`, which joins its two arguments using `mpl::insert_range`. The list monad can be used to represent ambiguity in pure code [28, 29].

IV.2.4 Reader

The reader monad combines functions operating on an immutable state. Monadic values are higher order functions taking the state as argument and returning some value. The monad itself doesn't deal with the state – it constructs functions operating on it. The result of a sequence of `bind`s is a function that takes the state as its argument.

In C++ template metaprogramming higher order functions are implemented using metafunction classes, thus in template metaprogramming the monadic values of the Reader monad are metafunction classes. A tag, `reader` needs to be created to make Reader an instance of `monad`:

```
template <> struct monad<reader> : monad_defaults<reader> {
  typedef lambda_c<t, _, t> return_;

  typedef
    lambda_c<a, f, r,
      lazy<
        mpl::apply<
          mpl::apply<
            already_lazy<f>,
            lazy_protect_args<mpl::apply<a, r>>
          >,
          already_lazy<r>>>>
    bind;
};
```

`return_` creates a constant function – regardless of the state it always returns `return_`'s argument. It is implemented using currying: `return_` is a metafunction class taking two arguments. When only one is provided, it returns a metafunction class taking the other argument.

The function created by `bind` takes a state as argument and calls `bind`'s first argument with it. The resulting value is used to construct a new `state -> value` function. The state is passed to this function to get the final result. `bind`'s implementation makes use of the currying provided by the lambda expressions (see section III.5.4). It is implemented as a lambda expression taking three arguments and can be used as a metafunction class taking only two arguments and returning a metafunction class taking one argument.

In the reader monad, monadic functions construct functions operating on the state based on the result of the previous function operating on the state. Thus, the execution of higher order code – code building functions operating on the state – is mixed with normal functions operating on the state.

IV.2.5 State

The State monad maintains a state like the Reader monad, but the monadic values are functions that can change the state: they are functions taking a state as an argument and returning a pair: a new state and a result.

The C++ template metaprogramming implementation of this monad is similar to the implementation of the Reader monad: higher order functions are represented by metafunction classes, pairs are implemented using pairs provided by Boost.MPL.

```
template <>
struct monad<state> : monad_defaults<state> {
  typedef lambda_c<t, s, mpl::pair<t, s>> return_;

  typedef
    lambda_c<a, f, s,
      eval_match_let_c<
        mpl::pair<t, u>, mpl::apply<a, s>,
        lazy<
          mpl::apply<
            lazy_protect_args<mpl::apply<f, t>>,
            already_lazy<u>
          >
        >
      >
    >
    bind;
};
```

`return_` creates a function returning `return_`'s argument and not changing the state. The function created by `bind` takes a state as argument and passes it to `bind`'s first argument. The resulting value is used to construct a new `state -> (value, state)` function. The new state is passed to this function to get the final result. The implementation of `bind` takes advantage of the currying support provided by the lambda expressions. It is implemented as a lambda expression taking three arguments and can be used as a metafunction class taking two arguments and returning a metafunction class taking one argument.

Given that C++ template metaprogramming is a pure functional language, there is no mutable global state. The State monad is a tool for simulating a mutating state for functions that need it.

IV.2.6 Writer

The Writer monad demonstrates the expressiveness of the typeclass implementation and an extension to tags used by Boost.MPL. The idea of the Writer monad is based on *monoid*s. In abstract algebra an object is called a monoid [48] if it meets the following requirements:

- It has an *associative binary operator*. That is, an operator, `*`, that satisfies the following equation: `a * (b * c) == (a * b) * c`.
- It has an *identity value*, `e`, that satisfies `a * e == a` and `e * a == a`

In Haskell, this concept is captured by the `Monoid` typeclass:

```
class Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mconcat :: [a] -> a
  mconcat = foldr mappend mempty
```

`mappend` implements the binary operation, `mempty` is the identity element. `mconcat` is a function concatenating the elements of a list using the binary operation. It has a default implementation that can be overridden by a more efficient algorithm for types where it is possible. This typeclass is implemented in template metaprogramming using the approach presented for implementing typeclasses (see section III.4). The full implementation can be found in Mpllibs [59].

The monadic values of the Writer monad are pairs: a value and a state. The states are expected to form a monoid, thus they have an associative operation that can merge a number of state values. The Writer monad collects the list of states while executing a sequence of `bind` calls and reduces them into one value using the binary operation of the monoid.

Creating a tag for the Writer monad is not as straight forward as it was for other monads, since the Writer monad expects a monoid instance as argument. The Haskell implementation expects the type of the state to be an instance of the `Monoid` typeclass. An extra argument is needed for the Writer monad: the tag of the monoid. It is provided by making the tag of the Writer monad a template class taking the tag of the monoid as argument.

```
template <class Monoid> struct writer {};
```

`writer` is an instance of the `monad` typeclass using partial specialisation [76]:

```
template <class Monoid>
struct monad<writer<Monoid>> : monad_defaults<writer<Monoid>> {
  // ...
};
```

It makes `writer` an instance of the `monad` typeclass independent of the monoid the Writer monad uses. Because of using typeclasses, the functions expected by `monad` is implemented in a generic way, without knowing which monoid is used:

```
template <class Monoid>
struct monad<writer<Monoid>> : monad_defaults<writer<Monoid>> {
  typedef lambda_c<t, mpl::pair<t, mempty<Monoid>>> return_;

  typedef
    lambda_c<a, f,
      lazy<
        mpl::pair<
          mpl::first<
            lazy_protect_args<mpl::apply<f, mpl::first<a>>>
          >,
          mappend<
            already_lazy<Monoid>,
            already_lazy<mpl::second<a>>,
            lazy_argument<
              mpl::second<
                lazy_protect_args<mpl::apply<f, mpl::first<a>>>
              >
            >
          >
        >
      >
    >
    bind;
};
```

This code snippet uses the `monoid` typeclass to refer to the monoid's operations. The functions `mempty` and `mappend` are wrappers of calling the corresponding functions of the `monoid<Monoid>` trait instance.

Using typeclasses made it possible to implement the Writer monad independently of the monoid used by the monad. The tag of the monoid is encoded in the tag of the Writer monad.

IV.3 Do notation

Haskell provides syntactic sugar for monads, called *do notation* [48]. In Haskell, a *do block* is associated with a monad and contains a number of monadic function calls and value bindings. Here is an example do block:

```
do
  r <- may_fail1 13
  may_fail2 r
```

This evaluates `may_fail1 13`, binds `r` to its result and evaluates `may_fail2 r`. Using this notation instead of calling `bind` directly makes the code easier to read and maintain. This section presents how to provide the do notation in template metaprogramming. A do block looks like the following:

```
do_<monad_tag,
  step1,
  // ...
  stepn>
```

`do_` is a template class, `monad_tag` is the `tag` identifying the monad. This has to be passed to the `return_` and `bind` functions. The rest of the arguments are the steps of the do block. Each step is either a nullary metafunction returning a monadic value or a binding of an expression to a name. Steps are always syntaxes. A binding is expressed by the following structure:

```
syntax<set<name, step>>
```

`step` is a nullary metafunction returning a monadic value, `name` is a variable. The binding binds the result of `step` to this name. The bound name can be used in the steps of the do block following the binding. Here is the example using the `may_fail` functions implemented using `do_`:

```
do_<exception_tag,
  syntax<set<r, may_fail1<mpl::int_<13>>>,
  syntax<may_fail2<r>>
>
```

This implementation uses `set` to bind the result of calling the `may_fail1` metafunction to a variable, `r`. This result is passed to `may_fail2` using the variable `r`.

Similarly to `let` and `let_c`, a `do_c` template can be provided to offer an easier way of writing do blocks. The implementation details are not presented here.

IV.3.1 Implementation of the do notation

Do blocks can be transformed into a sequence of `bind` and `bind_` calls without changing their meaning. This process is called the *desugaring of the do block* and is presented in detail in [48]. The solution presented here follows this process. `do_` is implemented the following way:

- `do_<monad_tag, syntax<step>>` evaluates `step`.
- `do_<monad_tag,syntax<step1>,step2,...,stepn>` evaluates `bind_<monad_tag, step1, do_<monad_tag, step2, ..., stepn>>`.
- `do_<monad_tag,syntax<set<name,exp>>,step2,...,stepn>` evaluates `bind<monad_tag, exp, lambda<name, syntax<do_<step2, ..., stepn>>>>`.

Following these rules, the example:

```
do_<exception_tag,
  syntax<set<r, may_fail1<mpl::int_<13>>>,
  syntax<may_fail2<r>>
>
```

is transformed into

```
bind<
  exception_tag,
  may_fail1<mpl::int_<13>>,
  lambda_c<r, may_fail2<r>>
>
```

When a do block is evaluated, it is transformed to a nullary metafunction like the example above and gets evaluated. The transformation happens when the do block is evaluated, thus do blocks that are not evaluated are never transformed.

IV.3.2 Using return_ in do blocks

The standard tool for creating monadic values from non-monadic ones is `return_`. It takes the `tag` of the monad and the non-monadic value and returns a monadic value. The following example demonstrates how to use it in do blocks:

```
do_<monad_tag,
  syntax<return_<monad_tag, mpl::int_<13>>>
>
```

The problem with this solution is that every time `return_` is used inside the do block, the `tag` of the monad has to be passed to it, as well. It requires specifying the `tag` of the monad at multiple places, which is more work to be done for the developer and more possibilities to make mistakes, leading to bugs that are difficult to find. There is a better solution that makes the do block deduce the first argument of `return_`. A new template class, `do_return` is provided to simplify this. Here is the previous example using this new tool:

```
do_<monad_tag,
  syntax<do_return<mpl::int_<13>>>
>
```

This example specifies the tag of the monad only once for the `do_` block, and it is not repeated when the `return` operation is used. It can be easily implemented by `do_<monad_tag, syntax<do_return<exp>, step2,...,stepn>` evaluating `do_<monad_tag,syntax<return_<monad_tag,exp>>,step2,...,stepn>`. A reference implementation can be found in [59].

With this, the implementation of do blocks is complete. As in Haskell, using do blocks make the source code easier to read and understand in C++ template metaprograms as well.

IV.3.3 List comprehension

List comprehension [48] is a syntactic sugar for creating lists from other lists. A number of languages, such as Haskell [48], Python [35], Erlang [6, 7] etc. support it. For example, assuming that an `is_relative_prime` function is available, here is how to get the list of relative primes in the range of [1..100] in Haskell:

```
[(i, j) | i <- [1..100], j <- [1..100], is_relative_prime i j]
```

The above code snippet evaluates `is_relative_prime` for every `i` and `j` pairs in the range [1..100] and collects the `(i, j)` pairs that are relative primes into a list. Here is how to do it using the list monad and the do notation [26]:

```
do
  i <- [1..100]
  j <- [1..100]
  guard $ is_relative_prime i j
  (i, j)
```

This code snippet does the same as the previous one, but it is based on the do notation and the List monad. The do notation is available in template metaprogramming. Here is how to use it to express the above list comprehension in template metaprogramming:

```
do_c<list_tag,
  set<i, mpl::range_c<int, 1, 101>>,
  set<j, mpl::range_c<int, 1, 101>>,
  guard<is_relative_prime<i, j>>,
  mpl::pair<i, j>
>
```

The above code snippet implements the same functionality as the previous Haskell version, but it is implemented using the do blocks presented earlier. Based on the approach for turning do blocks into `bind` calls presented in section IV.3.1, the above code block is turned into the following expression:

```
bind<list_tag,
  mpl::range_c<int, 1, 101>,
  lambda_c<i, do_c<list_tag,
    set<j, mpl::range_c<int, 1, 101>>,
    guard<is_relative_prime<i, j>>,
    mpl::pair<i, j>
  >>
>
```

The above code snippet does the desugaring of the first line. The `bind` operation of the list monad applies the monadic action – in this case the rest of the original do block – on every element of the list and concatenates the results. Thus, the rest of the do block is evaluated for every `i` in the range [1..100] and the results are concatenated. The same applies for `j`, thus the last two lines of the do block are evaluated for every `i` and `j` pairs. Based on the desugaring rules, the following expression is evaluated:

```
bind<list_tag,
  mpl::range_c<int, 1, 101>,
  lambda_c<i, bind<list_tag,
    mpl::range_c<int, 1, 101>,
    lambda_c<j, bind_<list_tag,
      guard<is_relative_prime<i, j>>,
      mpl::pair<i, j>
    >>
  >>
>
```

The above code snippet shows the original do block after desugaring. It evaluates the `bind_<...>` sub-expression for every `i` and `j` pair. The second and third arguments of `bind_` evaluate to lists. The result of this operation is the third argument of `bind_` repeated as many times as many arguments the other list, the second argument of `bind_` has.

In order to keep only the relative primes, `guard<is_relative_prime<i, j>>` has to return a list with one element when `i` and `j` are relative primes and an empty list otherwise. This is easy to implement:

```
template <bool C>
struct guard_c : mpl::list<void> {};

template <>
struct guard_c<false> : mpl::list<> {};

template <class C>
struct guard : guard_c<C::type::value> {};
```

The `guard_c` function returns either a one element or an empty list depending on its boolean literal argument. The `guard` function takes a thunk evaluating to a wrapped boolean, evaluates it, unwraps the result and instantiates `guard_c` using it.

By introducing the `MonadPlus` typeclass, the `guard` function can be generalised and implemented for other monads as well. This is out of the scope for this dissertation. This section has presented how using do blocks and the list monad make it possible to provide list comprehension in template metaprogramming.

IV.4 Exception handling in metaprograms

Two monads, Maybe and Either, have been presented targeting error handling in pure code. This section presents a third monad called the Exception monad, that supports error handling and extends this to simulate exception handling in C++ template metaprograms.

The Exception monad in C++ template metaprogramming treats every value as a monadic value. This wouldn't be possible in Haskell, since that language uses the type system to define the monadic values. The following code snippet defines a special data-constructor for representing errors:

```
template <class Detail>
struct exception;
```

`exception` values contain details about the error, this is what the `Detail` argument represents. The exception monad follows the same logic as the Either monad, treating `exception` values as `left` and other values as `right` values. Thus, `return_` is the identity function, `bind` implements error propagation.

The Exception monad stops the further execution of the sequence of `bind`s when an exception is thrown and propagates the error to the caller, who can either process this error information or propagate it further. This is how exceptions behave in runtime C++ [72].

Compile-time exception handling is presented using the `min` template metafunction as an example: it takes two arguments and returns the smaller one. It uses another metafunction, `less`, to decide which is the smaller argument. `min` is implemented in Boost.MPL [25] the following way:

```
template <class A, class B>
struct min :
  if_<less<A, B>, A, B>
{};
```

Let's assume that `less` is implemented in a way that is returns an instance of `exception` when its arguments can not be compared. When this happens, the first argument of `if_` is an exception instead of a logical value. The body of `min` is a template metaprogramming expression. A sub-expression of it, `less<A, B>`, calls another metafunction, `less`. When a sub-expression of an expression returns an exception, the exception propagation logic should stop the evaluation of the entire expression and make the exception the result of the expression (propagate the exception). In order to do this, the above example needs to be turned into monadic code:

```
template <class A, class B>
struct min :
  bind<exception, less<A, B>, lambda_c<t, if_<t, A, B>>>
{};
```

This code snippet evaluates the original expression in two steps: first it evaluates the sub-expression that may return an exception, then it evaluates the rest of the expression. The two steps are connected by `bind`.

Turning every template-metaprogramming expression into monadic code is a tedious and error prone process. It makes the code extremely difficult to read and maintain. This section presents how to implement a small embedded language, that resembles runtime exception handling in C++ template metaprogramming. This language allows the developer to use *try* and *catch* blocks in template metaprograms. These blocks are automatically translated into monadic code presented above. The embedded language is implemented using the C++ standard, it doesn't require any additional tools.

Compile-time try blocks are provided in template metaprogramming, which are template classes taking at least one argument: a `syntax`, which is the expression to evaluate a number of catch blocks. The `try` blocks turn the syntax into a monadic expression before evaluating it. The following implementation of `min` uses these compile-time try blocks:

```
template <class A, class B>
struct min :
  try_<
    syntax<if_<less<A, B>, A, B>>
  >
{};
```

This solution wraps the body of the original `min` implementation. `try_` is a template class taking a nullary metafunction as argument. It transforms this nullary metafunction wrapped by `syntax` into a series of `bind_` calls:

- When the nullary metafunction is a class that is not a template instance, it remains as it is.
- When the nullary metafunction is an instance of a template class, it is transformed into a series of `bind_` calls. An instance of the `f` template class with `T1` ... `Tn` arguments, `f<T1, ..., Tn>`, is transformed into the following:

```
struct t1_; typedef var<t1_> t1;
// ...
struct tn_; typedef var<tn_> tn;

bind<exception_tag, T1, lambda_c<t1,
 bind<exception_tag, T2, lambda_c<t2, /* ... */
  bind<exception_tag, Tn, lambda_c<tn,
    f<t1, /*...*/, tn>
  >>
 /* ... */ >>
>>
```

This transformation ensures that when a sub-expression of the syntax throws an exception, the exception is not passed to the function taking that value as an argument but is propagated out of the entire expression. Using these try blocks the exceptions can be propagated in the sequence of function calls, similarly to stack unwinding in runtime code. This transformation is similar to the logic of desugaring *do blocks* in Haskell [48]

Instances of the `try_` template class provide a nested type called `type`, that is a `typedef` of the result of the monadic calculation. Thus, `try_` can be used as a metafunction. Similarly to runtime execution, exceptions are either handled at some point or they are propagated out of the entire metaprogram and break the evaluation of it. Exceptions thrown at runtime are handled by *catch* blocks. Catch blocks filter the exceptions by predicates taking the exception as argument. A try block in addition to the original syntax can have any number of – including zero – catch blocks as arguments. When any of the catch blocks handles the exception, the result of evaluating the try block is the value returned by the handler code. When none of the catch blocks catches the exception, the result of the try block is the exception itself. An additional template class is introduced to represent a catch block:

```
template <class Name, class Pred, class Body>
struct catch_;
```

Instances of the above template class represent catch blocks. The `Name` argument is the variable used to refer to the exception, `Pred` is a syntax that evaluates to a boolean value. This expression is evaluated to determine if this catch block handles the exception or not. When it evaluates to true, then `Body` is evaluated, which is also a syntax. The result of `Body` is the result of the `try_` block. For example:

```
template <class A, class B>
struct min :
  try_<
    syntax<if_<less<A, B>, A, B>>,
    catch_<e, syntax<boost::is_same<e, division_by_zero>>,
      syntax<A>>,
    catch_<e, syntax<mpl::true_>, syntax<B>>,
    catch_<e, syntax<mpl::true_>, syntax<mpl::int_<13>>>
  >
{};
```

This code snippet uses three `catch_` blocks. All of them refers to the exception as `e`. When the body of `try_` does not return an exception, the `catch_` blocks are ignored. Otherwise the predicates of the `catch_` blocks are evaluated in order and the first one that evaluates to true is selected. This means, that the last one is never selected, as the predicate of the second `catch_` block always evaluates to true. When a `division_by_zero` exception is thrown, the `try_` block evaluates to `A`, when another exception is thrown, it evaluates to `B`.

In runtime C++, functions that are not prepared to handle exceptions can be called from `try` blocks without any further syntactic elements. When using monads, non-monadic operations need to be *lifted* [48] into the monad. But for the exception monad every class is a monadic value: instances of the `exception` template class are exceptions, other classes are values representing a successful computation. Because of this, there is no lifting when using compile-time exceptions, which makes it more like exceptions in runtime C++ code.

IV.4.1 Implementation of exception handling

The `try_` template does two things: turns a syntax into monadic code and supports catch blocks. This section presents how to implement these things.

Turning a syntax into a monadic expression

Given a syntax wrapping an angle-bracket expression, it needs to be turned into a monadic expression using the exception monad to ensure that exceptions are propagated properly from every sub-expression. For example the following syntax

```
syntax<mpl::if_<less<A, B>, A, B>>
```

should be turned into the following:

```
bind<exception_tag,
  bind<exception_tag, A,
    lambda_c<v1, bind<exception_tag, B,
      lambda_c<v2, less<v1, v2>>
    >>
  >,
  lambda_c<v1, bind<exception_tag, A,
    lambda_c<v2, bind<exception_tag, B,
      lambda_c<v3, mpl::if_<v1, v2, v3>>
    >>
  >>
>
```

This code snippet evaluates every sub-expression of the original one individually and connects the results with `bind` and lambda expressions. This guarantees that when a sub-expression returns an exception, the evaluation of the entire expression stops and the exception is propagated out.

This is implemented as a template metafunction taking the syntax as input and evaluating it as a monadic expression. Given that monads enforce the serial evaluation of the actions [49], this transformation enforces a fixed evaluation order for the wrapped expression. Here is the implementation of this metafunction, it is called `make_monadic`:

```
template <class S>
struct make_monadic;

template <class Exp>
struct make_monadic<syntax<Exp>> :
  Exp
{};
```

The above code snippet declares the metafunction `make_monadic` and implements the case when the angle-bracket expression to transform has no further sub-expressions and does not need to be transformed further. It can be evaluated.

Expressions with sub-expressions are instances of template classes. They are processed using template template class arguments and partial specialisation. For example expressions calling a metafunction taking two arguments are transformed by the following specialisation:

```
struct v1_; typedef var<v1_> v1;
struct v2_; typedef var<v2_> v2;

template <template <class, class> class T, class T1, class T2>
struct make_monadic<syntax<T<T1, T2>>> :
  bind<exception_tag, make_monadic<syntax<T1>>,
    lambda_c<v1,
      bind<exception_tag, make_monadic<syntax<T2>>,
        lambda_c<v2, T<v1, v2>>
      >>>
{};
```

This code snippet specialises `make_monadic` for expressions calling a metafunction `T` taking two arguments, `T1` and `T2`. The arguments are evaluated from left to right and the function call is evaluated after that. The different steps are connected using `bind`. Preparing `make_monadic` for metafunction calls with different arities can be implemented in a similar way, the details are not presented here. A reference implementation can be found in [59].

This metafunction needs to be prepared for treating boxed values as special ones and not to evaluate them, similarly to `lazy`. The details of this are not presented here.

Implementing catch blocks

`try_` uses `make_monadic` to evaluate the syntax representing the body of the try block and to propagate exceptions out of it. Then it checks if there were any exceptions:

```
template <class Body, class C1, ..., class Cn>
struct try_ :
  eval_case_<make_monadic<Body>,
    matches_c<exception<e>,
      handle_exception<e, mpl::vector<C1, ..., Cn>>
    >,
    matches_c<e, e>
  >
{};
```

This code snippet uses `make_monadic` to evaluate `Body` and checks the result using `eval_case`. When it is an exception, it calls `handle_exception` with the exception that was thrown and a vector containing the catch cases. Otherwise it returns the result of the evaluation. The template metafunction `handle_exception` is implemented the following way:

```
template <class E, class Cs>
struct handle_exception :
  mpl::fold<
    Cs,
    mpl::pair<mpl::true_, exception<E>>,
    lambda_c<s, a,
      eval_case<s,
        matches_c<mpl::pair<mpl::true_, _>,
          handle_catch<e, a>
        >,
        matches_c<_, s>
      >>> {};
```

This code snippet iterates over the catch blocks using `mpl::fold`. The state used in the fold is a pair: the first element is a boolean indicating if the exception has been processed by a catch block, the second element is the exception when it has not been processed and the result of the catch block in case it has been processed. The lambda expression called for every catch block calls the `handle_catch` metafunction in case the exception has not been handled yet, otherwise returns the state unchanged. The `handle_catch` metafunction is implemented the following way:

```
template <class E, class C>
struct handle_catch;

template <class E, class Name, class Pred, class Body>
struct handle_catch<E, catch_<Name, Pred, Body>> :
  mpl::eval_if_<
    typename eval_let<Name, syntax<E>, Pred>::type,
    lazy<mpl::pair<
      mpl::true_,
      lazy_protect_args<eval_let<Name, syntax<E>, Body>>
    >>,
    mpl::pair<mpl::false_, exception<E>>
  > {};
```

This code snippet substitutes the variable `Name` in the predicate `Pred` and evaluates it using `eval_let`. When it evaluates to true, `Body`, the body of the catch block is evaluated the same way to get the result of the `try_` block and a pair of `mpl::true_` and this result is returned. Otherwise a pair of `mpl::false_` and the exception is returned. With this, the implementation of `try_` is complete.

IV.5 Summary

This chapter presented how to implement monads in C++ template metaprogramming. The implementation of a number of monad variations, such as the Maybe, Either, List, Reader, State and Writes monads were also presented. An approach has been presented for simulating the do notation of Haskell and it has been discussed, how to provide List comprehension in C++ template metaprogramming. This chapter has also presented how to simulate exception handling in template metaprogramming based on a monad instance, the Exception monad.

Thesis 2: I have developed a method for implementing monads and a Haskell-like do syntax in C++ template metaprogramming and evaluated how a number of different monad variations available in Haskell can be implemented using this method. Based on this I have developed a method for simulating exception handling in C++ template metaprograms. (chapter IV)

Thesis 2.1: I have developed a method for implementing monads in C++ template metaprogramming. (section IV.1)

Thesis 2.2: I have evaluated how a number of monads available in Haskell can be implemented using the approach presented in Thesis 2.1. (section IV.2)

Thesis 2.3: I have developed a method for implementing a Haskell-like do syntax in template metaprogramming. (section IV.3)

Thesis 2.4: I have developed a method for simulating exception handling in C++ template metaprogramming based on monads. (section IV.4)

Table IV.1: Related publications

	[61]	[64]	[66]	[67]
2.1		×		×
2.2	×	×		×
2.3	×	×	×	×
2.4	×	×		×

Chapter V

Parser generator library

The techniques discussed in chapters III and IV support building template metaprograms following the logic of Haskell. This chapter introduces a different approach for taking advantage of the similarities of the two languages. It presents how to implement parsers taking string literals and parsing them at compile-time. This makes it possible to provide a Haskell-like language for writing template metafunctions, parse them using template metaprograms and execute them as part of the same compilation process. This approach provides a readable syntax for C++ template metaprograms.

Implementing parsers as template metaprograms is useful for embedding domain specific languages in general, they are not limited to providing a better syntax for template metaprograms. Section V.2 presents use cases, such as a type safe `printf`.

V.1 Implementation of the library

This section presents how to implement compile-time parsers in C++ template metaprogramming, how to construct parsers and how to prepare them for error-handling. This approach is based on parser combinators [5] and the implementation is built on top of Boost.MPL.

V.1.1 Representing the input text

The approach presented in this chapter uses template metaprogramming to implement parsers. The input of the parsers are strings, thus they need to be represented as template metaprogramming values to be able to do the parsing. Boost.MPL provides a string implementation that can be used. The syntax of embedding strings in the source code is the following:

```
mpl::string<'Hell','o Wo','rld!'>
```

It is based on multi-character constants, which allow grouping four characters together. However, the embedded string has to be split into four character chunks which is not convenient and is difficult to work with. A macro taking a string literal as argument and expanding to the template metaprogramming representation of that string would be easier to use:

```
_S("Hello World!")
```

This section presents how to provide this macro by combining generalised constant expressions and preprocessor metaprograms. `"Hello World!"` is a string literal, which becomes a character array. Accessing a character of it, such as `"Hello World!"[2]` is a constant expression. A constant expression can be an argument of a template:

```
mpl::push_back<
  mpl::push_back<
    // ...
    mpl::push_back<
      mpl::string<>,
      mpl::char_<"Hello World!"[0]>
    >::type,
    // ...
    mpl::char_<"Hello World!"[10]>
  >::type,
  mpl::char_<"Hello World!"[11]>
>::type
```

The above code snippet appends each character of the string literal `"Hello World!"` to an empty template metaprogramming string one by one. It can be hidden by a macro. Using preprocessor metaprograms, such as `BOOST_PP_REPEAT` from Boost.Preprocessor [34] the above code can be generated from `_S("Hello World!")`. The problem is that the length of the string has to be known in advance, thus the `_S` macro works with a predefined string length only. Generating different code based on the length of the string literal in a preprocessor metaprogram is an open question, a fixed number of steps are generated and used in all cases. During the generated iterations the end of the string needs to be detected. A special metafunction that optionally appends a character to a string is defined:

```
template <class S, char C, bool EndOfString>
struct append_string : mpl::push_back<S, mpl::char_<C>> {};

template <class S, char C>
struct append_string<S, C, true> : S {};
```

This metafunction takes an extra argument, which is a boolean value indicating if the character really needs to be appended. To avoid over indexing the string literal, a `constexpr` function accessing a character of the string is needed:

```
template <int Len>
constexpr char str_at(const char (&s)[Len], int n)
{ return n >= Len ? 0 : s[n]; }
```

The above function takes a char array and an index as arguments. It receives the length of the array as a template argument and checks if the array has been over indexed. When it is over indexed, it returns the `'\0'` character. Since it is a `constexpr` function, its result can be used as a template argument. The following code snippet uses all the above to construct a template metaprogramming string from a string literal:

```
#define s "Hello World!"

append_string<
  append_string<
    // ...
    append_string<
      mpl::string<>,
      str_at(s, 0), (0 >= sizeof(s) - 1)
    >::type,
    // ...
    s[10], (10 >= sizeof(s) - 1)
  >::type,
  s[11], (11 >= sizeof(s) - 1)
>::type
```

The above code snippet uses `str_at` to access the characters, thus it is protected against over indexing the array. It uses `sizeof` to check the length of the string and `append_string` to append the characters. To each `append_string` call it passes a boolean value indicating if that character is a character of the string or just a 0 character coming from `str_at`.

The above code snippet can be automatically generated by a macro, where `s` is a macro argument. The number of appends generated by the macro sets an upper limit on the length of the string. The maximum value comes from a macro, such as `LIMIT_STRING_SIZE` which can be defined by the user. The implementation of this macro is not presented here. A reference implementation can be found in [59] and the implementation details are discussed in [62]. Using this, an `_S` macro can be implemented taking a string literal as argument and expanding to an expression that evaluates to the template metaprogramming representation of the string.

V.1.2 Representing source locations

A compiler has to be able to give good error messages to the developer when the text he compiles is invalid. To help the developer finding and fixing the bug, the error message should contain the location of the error in the invalid source code. The locations of the input text have to be represented in template metaprogramming. A location is a pair of integers representing line and column number. An algebraic data-type is created for it:

```
template <class Line, class Col>
struct source_position;
```

Metafunctions for querying and updating these values are easy to implement:

- `get_col` queries the column information.
- `get_line` queries the line information.
- `next_char` returns a new source position pointing to the next character of the same line.
- `next_line` returns a new source position pointing to the first character of the next line.

V.1.3 Building parsers

A parser is a template metafunction class taking the following arguments:

- The text to parse.
- Location information (line and column number) of the beginning of the input text in the entire input to parse. This argument is important, because the parser is given either the entire input text, or only a suffix of it. For proper error reporting, the parser has to be aware of the exact location of the parsed characters in the input text.

The return value of the function is one of the following:

- A tuple of some resulting value, the remaining text and the location information of the beginning of the remaining text. This tuple is returned when the parser was successful. The first element of the tuple, the resulting value can be any template metaprogramming value. It can be either a syntax tree, the result of the evaluation of the input text or anything else.
- A pair of some error description and a source location. This is returned when the parser failed to parse the input. The error description can be any template metaprogramming value.

The result of a successful parsing needs to be differentiated from a rejection. This is represented by an algebraic data-type with two constructors:

```
template <class Result, class Remaining, class Pos>
struct accepted;

template <class Msg, class Pos>
struct rejected;
```

The constructor `accepted` represents that a parser has accepted the input. `Result` is a template metaprogramming value representing the result of parsing. It depends on the parser and the accepted input. `Remaining` is the unprocessed suffix of the input, `Pos` is the position representing the position of the first character of `Remaining` in the original input.

To parse some text, the text itself and a source location pointing to the first character of the first line has to be passed to the parser. Since it is often used, this source position gets a custom name, `start`.

Basic parsers

Here are three simple parsers:

- `return_`, a parser that always accepts its input and consumes nothing from the input text.
- `fail`, a parser that always rejects its input.
- `one_char`, a parser that consumes the first character of its input. It rejects empty input.

The following code snippet implements `return_`:

```
template <class Result>
struct return_ : lambda_c<s, p, accepted<Result, s, p>> {};
```

`return_<R>` is a metafunction class, where `R` is the result the parser returns for any input. It is implemented using a lambda expression taking two arguments:

- `s`, the input text.
- `p`, the source position of the beginning of the input text.

The rest of the simple parsers, `fail` and `one_char` can be implemented in a similar way. A reference implementation can be found in [59].

Combining parsers

Based on the simple parsers, more complex ones are implemented by combining the simple ones together in many different ways. Parsers are combined by functions taking parsers as arguments and building new parsers from them. These functions are called parser combinators [5]. As an example a parser combinator taking a parser and a predicate to build a new parser is implemented. The new parser accepts an input if and only if the original parser accepts it and the predicate returns true for the result of the parser.

```
template <class Parser, class Pred, class ErrorMsg>
struct accept_when :
  lambda_c<s, p,
    eval_case< mpl::apply<Parser, s, p>,
      matches_c<rejected<m, q>, rejected<m, q>>,
      matches_c<accepted<r, m, q>,
        lazy<
          mpl::eval_if<
            lazy_protect_args<mpl::apply<Pred, r>>,
            already_lazy<accepted<r, m, q>>,
            already_lazy<rejected<ErrorMsg, p>>
          >
        >
      >
    >
  >
{};
```

`Parser` is the parser to extend, `Pred` is the predicate to check the result with and `ErrorMsg` is a class representing a meaningful error message the new parser returns when the predicate returns false. `accept_when<Parser, Pred, ErrorMsg>` is the new parser, implemented using `lambda_c`. It applies `Parser` on the input first and the result of this is checked using `eval_case`. When `Parser` rejects the input, the error message is returned. Otherwise the predicate is evaluated with the result of `Parser` as its argument. When it returns true, the result of `Parser` is returned. Otherwise an instance of `rejected` is returned, using `ErrorMsg` as the error message.

Accepting characters

The next parser combinator supports implementing a parser that expects one specific character. When the first character of the input is the expected one, the parser accepts the input and consumes that character, otherwise the parser rejects the input.

```
template <class C> struct literal_expected;

template <class C>
struct lit : accept_when<
  one_char,
  lambda_c<c, mpl::equal_to<C, c>>, literal_expected<C>
> {};
```

`C` is a boxed character, `lit<C>` is a parser accepting the input when its first character is `C`. `literal_expected` is a template class representing the error messages this parser fails with when it rejects the input. The parser is implemented using the `accept_when` parser combinator. It uses `one_char` as the base parser and a lambda expression comparing the parsed character to `C` as the predicate.

Choices

Ordered choice is implemented as a parser combinator as well. An ordered choice applies a number of parsers in order. The result is the result of the first parser that accepts the input. When all of the parsers reject the input, the combined parser rejects it as well. The implementation of this parser combinator is not presented here, a reference implementation (`one_of`) can be found in [59]. Using this combinator a parser expecting a digit character is implemented by combining a number of `lit` parsers. This new parser is called `digit`.

```
template <char C> struct lit_c : lit<mpl::char_<C>> {};

typedef
  one_of<
    lit_c<'0'>, lit_c<'1'>, lit_c<'2'>, lit_c<'3'>, lit_c<'4'>,
    lit_c<'5'>, lit_c<'6'>, lit_c<'7'>, lit_c<'8'>, lit_c<'9'>
  > digit;
```

`lit_c` simplifies the creation of `lit` parsers. By combining `lit_c<'0'> ... lit_c<'9'>` with `one_of`, `digit` accepts any character in the range `'0' - '9'` but nothing else.

Semantic actions

Parsers produce some result when they accept a prefix of the input. For example `lit` returns the accepted character as its result. In most cases this is not what the result of parsing an input should be. These results are transformed by functions taking the original result as their argument and producing the result users of the parser need. These functions are called *semantic actions* [43]. Semantic actions are implemented by parser combinators combining a parser and a transformation function transforming the result of the parser into some other value. A reference implementation (`transform`) can be found in [59]. The following code snippet presents how to turn `digit` into a parser that returns the value of the digit as the result of parsing:

```
struct char_to_int : tmp_value<char_to_int> {
  template <class C>
  struct apply : mpl::int_<C::type::value - '0'> {};
};

typedef transform<digit, char_to_int> digit_val;
```

This example defines a metafunction class, `char_to_int`, that converts a digit character into an integer value. The example builds a new parser, `digit_val` from digit using the parser combinator described above.

Repetition

The kleene star [2] is implemented as a parser combinator. It takes a parser as argument and applies it repeatedly as long as it accepts the input text. The list of successful parser applications is the result of applying the combined parser. Its implementation can be found in [59], it is called `any`.

When the underlying parser rejects the input for the first time, `any` still accepts it as zero matches. Another parser combinator, `any1`, can be provided, that treats zero matches as a failure. The following example demonstrates how to build a parser accepting natural numbers using it:

```
typedef
  lambda_c<l,
    mpl::fold<
      l, mpl::int_<0>,
      lambda_c<s,c, mpl::plus<mpl::times<mpl::int_<10>,s>,c>>
    >> calculate_int_value;

typedef transform<any1<digit_val>, calculate_int_value> int_;
```

This combinator builds a list of integers by parsing the digits one by one. It calculates the value of the parsed integer number from the digits using `calculate_int_value`.

The above example builds a parser that parses digits, collects them in a sequence and once the parsing has completed it iterates over this sequence using `mpl::fold` to calculate some final result. The sequence is constructed just to iterate over it once later. A more efficient way of implementing it is an approach, which does the recognition of the digits (implemented by `any1` in the above example) and the folding (implemented by `mpl::fold` in the above example) together. Once an element is available, it is processed immediately by the callback function of the folding part.

A parser combinator called `fold` provides this. It simulates an `mpl::fold` over the results of the repeated application of the wrapped parser instead of collecting those results in a sequence. A reference implementation of it can be found in [59]. It has a `foldl1` version, which requires that the wrapped parser can be applied at least once. There are parsers doing the folding in the reverse order – they are called `foldr` and `foldr1`. This is how to implement the above example using these new combinators:

```
typedef foldl1<
  digit_val, mpl::int_<0>,
  lambda_c<s, c, mpl::plus<mpl::times<mpl::int_<10>, s>, c>>
> int_;
```

This is a more compact and more efficient implementation of parsing integer values. It uses `foldl1` instead of `any1`. Instead of collecting the digits into a sequence and calculating the result later, it processes every digit immediately once it has been parsed.

The above example used `mpl::int_<0>` as the initial state but required at least one digit to be there. The above parser would more effective by using the value of that mandatory digit instead of the value `0` as the initial state for folding. Special versions of the folding parser combinators are provided for this purpose. The reference implementation provides the `foldlp` and `foldrp` parser combinators for this. They have the same signature as `foldl` and `foldr`, but they take a parser to get the initial value with as their first argument. This is how to implement the above example using them:

```
typedef foldlp<digit_val, digit_val,
  lambda_c<s, c, mpl::plus<mpl::times<mpl::int_<10>, s>, c>>
> int_;
```

This parser uses `digit_val` to parse the first digit, its parsing result as the initial state for folding and the parser `digit_val` repeatedly to get the rest of the digits.

Sequence

A parser combinator applying a list of parsers in order is implemented as well. When any of them fails, the combined parser fails as well and skips the remaining parsers. When all of them succeed, the result of parsing is the list of results. This combinator is available in [59] as `sequence`.

As an example for this, it can be used to parse a digit in brackets, for example `(1)`. Three parsers need to be combined using `sequence` to parse this: one parsing the open bracket, one parsing the digit and one parsing the closing bracket:

```
typedef
  sequence<lit_c<'('>, digit_val, lit_c<')'>> digit_in_bracket;
```

This parser accepts a digit in brackets. It returns a sequence of the open bracket character (the result of `lit_c<'('>`), the value of the digit (the result of `digit_val`) and the close bracket character (the result of `lit_c<')'>`). The result of parsing the middle element, `digit_val` should be the result of parsing of the entire bracket expression.

Given how often it is needed, the reference implementation provides a parser combinator called `middle_of`. It is a special version of sequence expecting exactly three elements. It accepts the input when the entire sequence is accepted, but it returns the result of the middle parser only. The following code snippet demonstrates how to implement the above example using it:

```
typedef
  middle_of<lit_c<'('>, digit_val, lit_c<')'>> digit_in_bracket;
```

This parser accepts a digit in brackets. Instead of returning the sequence of results as the result of parsing, it returns only the value of the digit. The reference implementation provides the `first_of` (and `last_of`) parser combinators as well, that are also special `sequence` parsers. They work with any (non-zero) number of elements and return the result of the first (last) parser.

Summary

This section presented parser combinators implementing sequences, choices and repetition. Using these tools, parsers for complex grammars can be constructed.

V.2 Applications

Being able to parse the content of string literals at compile time makes it possible to embed code snippets implemented in domain-specific languages. Here is a list of use cases for this:

- *Generate types.* Construct new types as the result of parsing a DSL script and pass them to metaprograms or instantiate them at runtime.
- *Generate optimised executable code.* C++ template metaprograms and parsers implemented using them generate executable code by combining small inline functions. This gives the compiler the opportunity to optimise the code.
- *Generate runtime objects.* Metaprograms generate code that initialises these objects during static initialisation.
- *Generate constant values.* The result of metaprograms and parsers implemented as metaprograms is a constant expression producing constant values.
- *Do compile-time assertions.* Metaprograms optionally break the compilation process based on some conditions. These conditions are implemented in a DSL and parsed at compile-time.

- *Generate template metaprograms.* Template metaprograms are generated as the result of parsing a DSL script. This makes it possible to implement easy to read languages for template metaprogramming, that work like scripts interpreted by the C++ compiler.

The following sections highlight areas where using compile-time parsers makes a significant difference.

V.2.1 Interface wrappers of libraries

Libraries have their own specific domains with their own common notations. The more the interface of the library follows that notation, the easier the experts of that domain can use it. Many C++ libraries [18, 43, 44, 13] try to follow the syntax of special problem domains by overloading C++ operators. Boost [33] provides the Proto [43] library making the development of such approaches easier. However, this approach has its constraints: all DSL expressions have to be valid C++ expressions as well.

Being able to parse DSL code snippets at compile-time makes it possible to provide an interface that follows the notation of the domain without being constrained by the syntax of C++ expressions.

As an example, one can look at Boost.Xpressive [44]. It provides an interface for building regular expressions. Assuming that the regular expression is available at compile-time, the user of the library can choose one of the following options:

- Embed the regular expression in the C++ code as a string literal. This is parsed at runtime – parsing has its own runtime cost and when the regular expression is invalid, it causes a runtime error, thus, the problem is not detected until runtime.

- Embed the regular expression in the C++ code as a C++ expression. Xpressive provides an interface for that and it follows the logic of regular expressions. However, the commonly used syntax had to be altered to make it a valid C++ expression, which makes it difficult for people not familiar with the syntax of Xpressive to read and understand it.

A third option, offered by compile-time parsers is embedding the regular expression as a string literal and parsing it at compile time to build the same structure one could build with the C++ expression-based interface. As an example here is how the regular expression `ab*c` can be implemented as a static (and therefore optimised) regular expression in Boost.Xpressive and how it can be implemented using a library wrapper which can be provided using the approach discussed in this chapter:

```
// Using Boost.Xpressive directly

boost::xpressive::as_xpr('a') >> *boost::xpressive::as_xpr('b')
>> boost::xpressive::as_xpr('c');


// Using a library wrapper built using the
// discussed techniques

REGEX("ab*c")
```

As the above example shows, parsing strings at compile-time makes it possible to provide simpler interface to existing libraries. People who are already familiar with regular expressions can understand the new interface without additional help given that it uses the common syntax while it takes time to learn the way regular expressions can be constructed using Boost.Xpressive. This interface wrapper for Boost.Xpressive is discussed in detail in [54].

V.2.2 Use-case: implementing a type-safe `printf` as a DSL

The C standard library has a function, `printf` for outputting formatted text. Its first parameter is a format string specifying how to format the rest of its arguments. When the number or the types of the arguments are invalid according to the format string, the runtime behaviour is undefined. This section presents how to use compile-time parsers to verify it at compile-time.

The syntax of the format string is an internal language inside C or C++. It is an embedded DSL and this section presents how to parse it using tools built for processing embedded DSLs. Using the original, non-type-safe version of `printf` looks like the following:

```
printf("%d + %d = %d\n", 11, 2, 13);
```

Using the type-safe `printf` presented in this section is similar to that:

```
safe::printf<_S("%d + %d = %d\n")>(11, 2, 13);
```

The type-safe `printf` is a template function taking the format string as a template argument and the rest of `printf`'s arguments as runtime arguments. When the number or types of the arguments are invalid according to the format string, it generates a compilation error. It calls the original, unsafe version of `printf` otherwise.

The application developer uses `safe::printf` in his code after including the DSL definition as a C++ header file. This definition contains the syntax of `safe::printf` in a DSL format. The header also includes an implementation of the parsing library presented earlier placed in separate header(s). When the user compiles his code, the C++ compiler executes the parser generator, which constructs the parser for `safe::printf`. This parser is immediately used (in the same compilation phase) to type check the arguments of `safe::printf`. The result is either executable code utilizing the standard `printf` or a compilation error reporting type mismatch.

The implementation of this template function for a format string with two placeholders is presented here. The rest of the overloads can be implemented in a similar way and the Boost.Preprocessor library [34] provides tools to automatically generate them.

```
template <class FormatString, class T1, class T2>
int safe_printf(T1 t1_, T2 t2_) {

  BOOST_STATIC_ASSERT((
    valid_printf<FormatString, mpl::list<T1, T2>>::type::value
  ));

  return
    printf(
      mpl::c_str<FormatString>::type::value,
      t1_,
      t2_
    );
}
```

It uses `mpl::c_str` to convert the compile-time string into a runtime one that is passed to the unsafe version `printf`. `BOOST_STATIC_ASSERT` is a macro taking a compile-time predicate as input and generating a compilation error when that predicate returns false. The predicate calls a template metafunction, `valid_printf`, and passes the format string and the list of argument types as arguments. `valid_printf` generates the list of expected argument types from the format string and compares it with the list of actual ones. When the two lists don't match, `valid_printf` returns false, thus the static assertion emits a compilation error. An expected argument is represented by an algebraic data-type with one constructor:

```
template <class LengthAvail, class PrecAvail, class T>
struct expected_arg;
```

This constructor has the following arguments:

- `LengthAvail`, a boolean value telling if a preceding integer argument describing the display length is expected.
- `PrecAvail`, a boolean value telling if a preceding integer argument describing the precision is expected.
- `T`, the expected type of the argument. This is described by another algebraic data-type, where each constructor represents a different type. It has the following constructors:

  ```
  struct expect_character;
  struct expect_double;
  // ...
  ```

As an example here is a format string and the list of expected arguments needed:

```
// format string
"%d + %.*d = %*u"

// list of expected arguments
using mpl::true_;
using mpl::false_;

mpl::list<
  expected_arg<false_, false_, expect_character>,       // %d
  expected_arg<false_, true_,  expect_signed_integer>,  // %.*d
  expected_arg<true_,  false_, expect_unsigned_integer> // %*u
>
```

Only the placeholders are important. The rest of the format string is ignored during type-checking.

`valid_printf` generates the list of expected argument types by parsing the format string. The parser is built using the parser combinator library. It takes the format string as input and produces the list of expected types as the result of parsing. Here is the grammar of `printf` format strings based on [65].

```
S               ::= CHARS (PARAM CHARS)*
PARAM           ::= '%' FLAG* WIDTH PRECISION FORMAT
FORMAT          ::= 'h' FORMAT_HFLAG  | 'l' FORMAT_LFLAG |
                    'L' FORMAT_LLFLAG | FORMAT_NO_FLAG
FORMAT_LLFLAG   ::= 'e'|'E'|'f'|'g'|'G'
FORMAT_LFLAG    ::= 'c'|'d'|'i'|'o'|'s'|'u'|'x'|'X'
FORMAT_HFLAG    ::= 'd'|'i'|'o'|'u'|'x'|'X'
FORMAT_NO_FLAG  ::= 'c'|'d'|'i'|'e'|'E'|'f'|'g'|'G'|
                    'o'|'s'|'u'|'x'|'X'|'p'|'n'|'%'
PRECISION       ::= '.' WIDTH | NONE
WIDTH           ::= INTEGER | '*' | NONE
INTEGER         ::= DIGIT+
DIGIT           ::= '0'|'1'|'2'|'3'|'4'|
                    '5'|'6'|'7'|'8'|'9'
FLAG            ::= '-'|'+'|' '|'#'|'0'
CHARS           ::= ('\' one_char | not ('%' | '\'))*
NONE            ::= epsilon
```

`CHARS` represents non-interpreted characters, `PARAM` represents one parameter to be substituted. The parser has to skip all non-interpreted characters, determine the type required by the PARAM parts and build the list of these types.

The parser is constructed based on the above grammar. `sequence` is used to implement `|`, `any` to implement `*`, `any1` to implement `+`, `except` to implement `not` and `return_` to implement `epsilon`. Using them, a parser can easily be constructed from the grammar: starting with basic parsers more and more complex parsers can be built using the combinators until it gets complex enough to parse the entire grammar. Parsing the `FORMAT_...` elements is simple. As an example, here is an implementation of `FORMAT_HFLAG`:

```
typedef
  one_of<
    always<lit_c<'d'>, expect_short_signed_integer>,
    always<lit_c<'i'>, expect_short_signed_integer>,
    always<lit_c<'o'>, expect_short_signed_integer>,
    always<lit_c<'u'>, expect_short_unsigned_integer>,
    always<lit_c<'x'>, expect_short_unsigned_integer>,
    always<lit_c<'X'>, expect_short_unsigned_integer>
  >
  format_h_flag;
```

The result of parsing such a formatting character is a place holder for an expected type. FORMAT_LFLAG, FORMAT_LLFLAG and FORMAT_NO_FLAG can be implemented in a similar way. Using these parsers, a parser for FORMAT parsing a format character with a flag is created:

```
typedef
  one_of<
    last_of<lit_c<'h'>, format_h_flag>,
    last_of<lit_c<'l'>, format_l_flag>,
    last_of<lit_c<'L'>, format_capital_l_flag>,
    format_no_flag
  >
  format;
```

It uses the appropriate format character parser based on the flag controlling the type of the printf argument. Every argument of one_of is a parser, which either succeeds completely or fails. last_of applies all of its arguments in sequence, expects all of them to succeed. The result of last_of is the result of the last sub-parser, thus the above implementation parses the format controlling flag ('h', 'l' or 'L'), throws the result away and calls the appropriate format_... parser. The lit_c parsers act like guards. The parser for WIDTH is implemented the following way:

```
typedef
  one_of<
    always<integer, mpl::false_>,
    always<lit_c<'*'>, mpl::true_>,
    return_<mpl::false_>
  > width;
```

The result of this parser is a boolean indicating if there should be an integer argument specifying the precision or not. Since this approach only cares about type checking the runtime arguments of printf, the value of the precision is ignored. PRECISION is implemented using WIDTH:

```
typedef
  one_of<
    last_of<it_c<'.'>, width>,
    return_<mpl::false_>
  > precision;
```

The result of this parser is a boolean similarly to WIDTH. FLAG parses a one character flag, its implementation is straightforward:

```
typedef
  one_of<
    lit_c<'-'>, lit_c<'+'>, lit_c<' '>, lit_c<'#'>, lit_c<'0'>
  > flag;
```

This parser returns the flag it has parsed. Since the goal is type-checking the arguments of `printf`, these flags can be safely skipped. `PARAM` is implemented based on these:

```
typedef
  last_of<
    lit_c<'%'>,
    any<flag>,
    transform<
      sequence<width, precision, format>,
      lambda_c<s,
        expected_arg<
          mpl::front<s>,
          mpl::at<s, mpl::int_<1>>,
          mpl::back<s>
        >
      >
    >
  >
  param;
```

`param` parses the entire description of a place holder. It expects a `%` character, then it parses the flags if there are any, the display length, the precision and the format character itself. It uses `sequence` to build a compile-time list of the results of the last three elements. This compile-time list is transformed into an `expected_arg` by a `transform` parser, thus this parser produces the elements needed at the end of parsing: the `expected_arg` values describing the expected arguments of `printf`.

In a format string the placeholders are separated by characters `printf` prints out as they are. Those characters are ignored. Sequences of those characters are parsed by `CHARS`:

```
typedef
  any_one_of<
    second_of<lit_c<'\\'>, one_char>,
    second_of<except<lit_c<'%'>, int>, one_char>
  > chars;
```

`except` is a parser combinator building a new parser that accepts everything the original one rejects. It is a look-ahead parser, since it doesn't consume any input. `any_one_of` is a combination of `any` and `one_of` to make parser definitions more compact. Having all these parsers the top-level parser of the `printf` grammar is defined:

```
typedef last_of<chars, any<first_of<param, chars>>> S;
```

This top-level parser ignores normal characters and collects the parsed placeholders – the `expected_arg` values – into a list. This list is the result of parsing and is used to validate the type of the runtime arguments.

To simplify the usage of the parsers, a convenience metafunction is provided taking care of applying a parser on an input and getting the final result. This is implemented the following way:

```
template <class P>
struct build_parser :
  lambda_c<s,
    eval_case<mpl::apply<P, s, start>,
      matches_c<accepted<r, _, _>, r>,
      matches_c<e, e>
    >> {};
```

`build_parser` takes the top-level parser as argument and builds a lambda function taking the input string as argument and producing the result of parsing, when it was successful and the error otherwise. Using it the top-level parser for the `printf` grammar is constructed the following way:

```
typedef build_parser<S> printf_parser;
```

The above definition builds a parser for the `printf` grammar. The following example demonstrates how to use it:

```
printf_parser::apply<_S("%d + %d = %d\n")>::type
```

A solution validating the arguments of `printf` at compile-time and emitting a compilation error when they are invalid has been presented. For this solution to work, the format string has to be available at compile-time, however, format strings are rarely constructed dynamically. Due to the extra validation at compile-time, this approach affects the speed of compilation but it has no extra runtime cost.

V.3 Building EDSLs for template metaprogramming

It is possible to construct template metaprograms as the result of parsing an embedded DSL code snippet. This section presents how to do it and how to build an easy to read language for C++ template metaprograms. Given the similarities between Haskell and C++ template metaprogramming [48, 37, 8] the language presented here is similar to Haskell.

As an example demonstrating what value this DSL adds to template metaprogramming let's take a look at the final, working version of the factorial function at the end of section III.1.1. This is `fact` implemented based on a widely used library, Boost.MPL:

```
template <class N> struct fact;

template <class N>
struct fact_impl :
  mpl::times<
    typename fact<
      typename mpl::minus<N, mpl::int_<1>>::type
    >::type, N>
{};

template <class N>
struct fact :
  mpl::eval_if<
    typename mpl::less<N, mpl::int_<1>>::type,
    mpl::int_<1>,
    fact_impl<N>
  >
{};
```

Basic arithmetic operations are available as template metafunctions and due to the lack of laziness a helper metafunction, `fact_impl` had to be introduced. The algorithm used to calculate factorial numbers is difficult to understand. The same function can be implemented using the DSL this chapter presents the following way:

```
"fact n ="
  " if n == 0"
    " then 1"
    " else n * fact (n-1)"
```

This implementation uses the Haskell-like syntax to implement `fact` and makes use of the compactness of Haskell to make it easy to understand what it is doing.

V.3.1 Parsing and building an AST

The following simple language is used as the starting point and it is extended later.

```
single_exp ::= int_token | name_token
```

This language can express simple expressions. An expression is either an integer value or the name of a variable or function. It is parsed into an *abstract syntax tree* [2] (AST), in which the following templates represent integer values and variable or function references:

```
template <class Val> struct ast_value;
template <class Name> struct ast_ref;
```

`ast_value` represents a value, `ast_ref` represents a reference to something. A parser for the above grammar is constructed:

```
typedef
  transform<int_token, lambda_c<x, ast_value<x>>::type>
  int_exp;

typedef
  transform<name_token, lambda_c<x, ast_ref<x>>::type>
  name_exp;

typedef one_of<int_exp, name_exp> single_exp;
```

The above code snippet defines a parser parsing integers (`int_exp`) and one parsing names (`name_exp`). The `int_token` and `name_token` parsers accepting an integer and an identifier can be easily implemented, the details are not presented here. The `_token` parsers return the parsed values which are turned into ASTs by using the `transform` combinator. Finally, the two parsers are combined by using the `one_of` combinator to accept either an integer or a name.

Let's extend the above language with function application with the following syntax: `<function> <arg1> <arg2> ...` where `<function>` is an expression - it can be the name of a function or an expression evaluating to a function (such expressions are presented later). The following grammar is used for function application:

```
application ::= single_exp+
```

The function and its arguments are represented as expressions. The first expression is expected to evaluate to a function accepting at least as many arguments as the expression applies on it. Accepting more is not a problem, because this language will support currying. Not meeting this requirement causes an error during the execution of the compiled metaprogram. This language has no type system that could detect these errors ahead of execution. Applications are represented in the AST by instances of the following template class:

```
template <class F, class Arg>
struct ast_application;
```

This AST element represents applying one argument on a function. Following the logic of currying, arguments are applied one by one. Thus, parsing the expression `f 1 2 3` gives the following AST:

```
ast_application<
  ast_application<
    ast_application<
      ast_ref<mpl::string<'f'>>,
      ast_value<int_<1>>
    >,
    ast_value<int_<2>>
  >,
  ast_value<int_<3>>
>
```

The above AST applies the arguments on the function `f` one by one. The parser for `application` can be constructed from `single_exp` by using the `any1` and `transform` parser combinators. The `fold` parser combinator can be used to implement it in an efficient way.

V.3.2 Binding references

After building the AST, references need to be bound in it to metafunctions and values (which are classes in template metaprogramming). The symbol table is represented by an `mpl::map`. The keys are the names, the values are the values (or metafunctions classes) to bind to. A metafunction, `bind` is implemented, that takes an AST and a symbol table as arguments and does the binding.

The result of binding is an expression, where all references are resolved. `bind` could evaluate it immediately, but doing it would lead to an eager evaluation strategy for the Haskell-like language. To make the language more like Haskell, `bind` returns the expression as a thunk, that can be evaluated at a later point in time. These thunks are constructed by combining the following templates:

```
template <class V>
struct lazy_value { typedef V type; };

template <class F, class Arg>
struct lazy_application :
  F::type::template apply<Arg>::type {};
```

Two template classes are used: one representing values and one representing function applications. There is no template representing references, since the thunks are constructed after doing the binding, during which all references are expected to be resolved. Invalid references are interpreted as errors.

Both of the above template classes calculate the result of the expression when they are evaluated by accessing their `::type`. Since values are assumed to be evaluated lazily, when an argument is applied on a function by `lazy_application`, the function may be the result of an unevaluated expression, that needs to be evaluated. This is why `lazy_application` uses the `::apply<...>` of `F::type`. The following code snippet presents how to implement `bind` using these template classes:

```
template <class AST, class Sym>
struct bind;

template <class V, class Sym>
struct bind<ast_value<V>, Sym> {
  typedef lazy_value<V> type;
};

template <class F, class A, class Sym>
struct bind<ast_application<F, A>, Sym> {
  typedef
    lazy_application<
      typename bind<F, Sym>::type,
      typename bind<A, Sym>::type
    > type;
};
```

The above code snippet defines `bind` for constant values and function applications. Constant values are wrapped with `lazy_value` to make sure that they can be used by lazy metafunctions. In case of function applications, the binding is done recursively on the expression constructing the function (`F`) and the expression used as argument (`A`). The result of these two bindings are used to construct a `lazy_application` value. The following specialisation of `bind` implements the binding of references:

```
template <class Name, class Sym>
struct bind<ast_ref<Name>, Sym> : mpl::at<Sym, Name> {};
```

This code snippet gets the value referenced by the reference from the symbol table, `Sym` using `mpl::at`.

V.3.3 Constructing the symbol table

The symbol table can be constructed using the constructors of `mpl::map`, however, it makes it difficult to read the code later. This section presents a small DSL for constructing the symbol table. Here is an example demonstrating how the DSL can be used:

```
template <class T> struct f;

typedef meta_hs
  ::import<_S("some_value"), mpl::int_<13>>::type
  ::import1<_S("f"), f>::type
  ::import2<_S("plus"), mpl::plus>::type
sample_map;
```

`meta_hs` is a class with nested template classes called `import`, `import1`, `import2`, etc. They take a name as a template metaprogramming string and the referenced entity as arguments. `import` adds values, while `import1`, `import2`, etc add metafunctions to the symbol table. The following code snippet presents how to implement this class:

```
template <class Sym>
struct meta_hs_builder : tmp_value<meta_hs_builder> {
  template <class Name, class V>
  struct import :
    meta_hs_builder<
      typename mpl::insert<Sym, mpl::pair<Name, V>>::type
    > {};
```

```
  tempkate <class Name, template <class> class F>
  struct import1<Name, F> :
    meta_hs_builder<
      typename mpl::insert<
        Sym,
        mpl::pair<Name, curry1<F>>
      >::type
    > {};

  // ...
};

typedef meta_hs_builder<mpl::map<>> meta_hs;
```

This code snippet uses the `meta_hs_builder` template class to construct the symbol table. The partially constructed symbol table is the template argument of the class. `meta_hs` is this template class instantiated with the empty map. The `import` templates are nested templates of `meta_hs_builder`, they instantiate `meta_hs_builder` with a map containing the new element. The `import` macros importing functions use the `curry` templates presented in section III.2. The Boost.Preprocessor library [34] provides tools for automatically generating the `import` templates. The details are not presented here. A reference implementation can be found in [59].

V.3.4 Adding functions written in the new language to the symbol table

Using `meta_hs` regular metafunctions can be added to the symbol table. This section extends it to make it possible to implement functions in the new language and bind them to names in the `mpl::map`. The resulting syntax will be the following:

```
typedef meta_hs
  ::define<_S("f x y = ...")>::type
sample_map;
```

This definition defines a function using the new language and binds it to the name `f`. First the grammar of the new language needs to be extended to handle these definitions:

```
definition ::= name_token+ define_token application
```

The definition of a function begins with the name of the function followed by the names of the arguments. This is the `name_token+` part of the above rule. It is followed by `define_token`, which expects an `=` character. An expression (`application`) describes the body of the function. Lambda abstractions are introduced in the AST to describe function definitions:

```
template <class F, class ArgName> struct ast_lambda;
```

It describes one lambda argument. A function expecting multiple arguments is represented by nested lambda abstractions. The `bind` function needs to be prepared for handling these lambda abstractions. The body of a lambda abstraction may refer to the argument of the lambda abstraction and those references need to be bound to the value the lambda abstraction is called with. But this value is not available when the binding of the lambda abstraction itself happens. `bind` *delays* the binding of the body until the value of the argument is available:

```
template <class F, class ArgName, class Sym>
struct bind<ast_lambda<F, ArgName>, Sym> {
  typedef bind type;

  template <class ArgValue>
  struct apply :
    bind<F,
      typename mpl::insert<
        Sym, mpl::pair<ArgName, ArgValue>
      >::type
    >::type {};
};
```

This overload of `bind` implements the binding of a lambda abstraction. The result of this binding is a metafunction class accepting one argument. When this metafunction class is called, it binds the value it was called with to the name of the argument in the symbol table, that was used when the binding of the lambda abstraction happened – `mpl::insert` is used for this. The binding of the body of the lambda expression happens using this new symbol table and the resulting thunk is evaluated immediately. The result of the metafunction class call is the result of this evaluation.

Function definitions represented by the `definition` element of the grammar are represented as `ast_lambda` elements in the AST. They are parsed by the following parser:

```
typedef
  sequence<
    name_token,
    foldrp<
      name_token, last_of<define_token, application>,
      lambda_c<a, f, ast_lambda<f, a>>
    >> accept_definition;
```

This implementation uses the `sequence` combinator to parse the name of the function using the `name_token` parser and the remaining part of the definition: the list of arguments and the body of the function. This remaining part is parsed using the `foldrp` combinator. It folds over the arguments of the function, each of them is parsed using the `name_token` parser. The initial state of folding is the body of the function. It is parsed using the `application` parser. This parser returns the AST of the body as the result of parsing. The state during folding is the AST of the already processed part. Each iteration adds a new argument to the function as the folding iterates over the formal argument list. It means, that the callback function used during folding needs to combine the old state with the current variable in a lambda abstraction – that is what the lambda expression used there does.

The result of the `accept_definition` parser is a two element sequence. The first element is the name of the function, the second one is the function definition represented as a lambda abstraction. Let's extend `meta_hs` to support the `::define` elements for implementing functions using these lambda abstractions:

```
typedef meta_hs
  ::import1<_S("add"), mpl::plus>::type
  ::define<_S("f x y = add x y")>::type
sample_map;
```

The above example imports `mpl::plus` with the name `add` and then defines the function `f` with two arguments, `x` and `y`. The body of it evaluates `add x y`. `define` uses the parser presented above to parse the definition of `f`. The result of parsing is the name `f` and an AST describing the arguments and the body of the function as a lambda abstraction. This AST needs to be bound and the result of that binding is associated with the name `f` in the symbol table. This binding needs a symbol table – which has to be the symbol table constructed *before* the definition of `f`. In this case it means, that this symbol table contains only the `add` function. As after this binding step `f` is added to the symbol table, further definitions can use it.

V.3.5 Recursive functions

The above approach does not allow defining recursive functions, since a function is added to the symbol table after its definition is fully parsed and bound. To support recursive functions and make it possible to refer to functions defined later, the symbol table should store the abstract syntax trees instead of the result of the bindings. The binding is done when an element of the symbol table is used in a calculation. By then all functions are already in the symbol table, thus functions can refer to themselves or other functions defined later.

When ASTs are stored in the symbol table, `import` should turn the imported entities into ASTs to be able to add them to the `mpl::map` representing the symbol table. As imported things should be ignored during binding, they are represented as `ast_value` items in the AST.

When the symbol table stores ASTs, then the result of a symbol table lookup is an AST that needs to be bound. For example:

```
meta_hs
  ::define<_S("a = b")>::type
  ::define<_S("b = 2")>::type
```

The above code snippet defines a symbol table with two elements: `a` referring to `b` and `b` referring to the value `2`. When the binding of the expression `a` happens, `bind` gets the AST `a` refers to, which is `b`. As this is an AST, it also needs binding. What `bind` does in this situation is *recursive binding* of these expressions: after doing the lookup of an AST it needs to be bound with the same symbol table. Thus, after getting the AST `b` – this is what `a` refers to – it also needs to be bound in the same symbol table. As in this symbol table `b` refers to `2`, the result of this is the AST `2`. As this is an AST, this needs binding as well, which results in the value `mpl::int_<2>`. The following specialisation of `bind` implements this:

```
template <class Name, class Sym>
struct bind<ast_ref<Name>, Sym> :
  bind<typename mpl::at<Sym, Name>::type, Sym> {};
```

This specialisation does the lookup of the reference using `mpl::at` which returns the AST the reference refers to in the symbol table. The binding of this AST happens by calling `bind` recursively using the same symbol table, `Sym`.

Recursive binding may lead to problems when it is used together with lambda abstractions. To demonstrate this, let's extend the above example symbol table with a lambda abstraction definition:

```
meta_hs ::define<_S("a = b")>::type
        ::define<_S("b = 2")>::type
        ::define<_S("f b = a")>::type
```

The above code snippet defines a symbol table with three elements: `a` referring to `b`, `b` referring to `2` and `f`, a lambda abstraction taking one argument, ignoring it and returning `a`. When the expression `f 1` is evaluated using the above symbol table, the argument `1` is applied on the lambda abstraction `f`. Based on the way lambda abstractions are evaluated, it builds a new symbol table in which `b`, the argument of the lambda abstraction refers to the argument value, `1`. The binding of the body of the lambda – `a` – happens using this symbol table. `a` refers to the syntax tree `b` and since `bind` should be doing recursive binding using the same symbol table, it should do the binding of `b`. But the lambda abstraction has changed what the name `b` refers to – in this case it refers to `1`, the argument the lambda abstraction was called with.

The above problem can be avoided by doing the recursive binding of the referred ASTs using the *original* symbol table without the overrides done by the lambda abstractions. One way of implementing it is giving `bind` two symbol tables: one that may be overridden by the lambdas and the original one that is never overridden:

```
template <class AST, class Sym, class OriginalSym>
struct bind;
```

This version of `bind` takes two symbol tables: one of them is `Sym` which is the same as `Sym` was in the previous version of `bind`, the symbol table which may be overridden by lambda abstractions. The other one is `OriginalSym`, which is the top-level symbol table without any overrides done by the lambda abstractions. The specialisations of `bind` presented so far are extended to this new signature by using `Sym` for the lookup, overriding it in the lambda abstractions and passing `OriginalSym` unchanged further when doing recursive `bind` calls.

Let's introduce a new AST element representing the root of an AST in the symbol table:

```
template <class E> struct ast_root;
```

As this is the root of an element in the symbol table, the overrides done by the lambda abstractions to the symbol table to do the binding in need to be ignored and the original symbol table needs to be used as the symbol table:

```
template <class E, class Sym, class OriginalSym>
struct bind<ast_root<E>, Sym, OriginalSym> :
  bind<E, OriginalSym, OriginalSym> {};
```

This implementation of `bind` calls `bind` recursively to do the binding of the wrapped AST, but replaces the potentially overridden symbol table `Sym` with the original symbol table, `OriginalSym`.

V.3.6 Exporting functions

It is possible to build simple functions using the new language and construct more complicated ones from them and regular metafunctions defined outside of the `meta_hs` block using these techniques. To be able to construct metafunctions that can be used outside of the `meta_hs` block, functions defined in a `meta_hs` block need to be exported. Let's introduce the following syntax for this:

```
typedef meta_hs ::define<_S("id x = x")>::type
                ::get<_S("id")>::type id;
```

`get<_S("...")>` exports a metafunction. Its result is a metafunction class that behaves the same way as any other metafunction class:

```
typedef id::apply<int>::type also_int;
```

`define` elements bind abstract syntax trees to names. Metafunctions are constructed when the binding happens. Thus, `get` has to do the binding of the AST the name is mapped to. The mapping `get` uses is the entire mapping `meta_hs` has constructed, thus functions can be referenced before they are defined – names are resolved when the exporting happens. For example, the following works:

```
typedef meta_hs ::define<_S("f x = g x")>::type
                ::define<_S("g x = x")>::type
                ::get<_S("f")>::type f;
```

In the above code snippet `f` uses `g`, which is defined later, but it works, since `f` is exported after `g` has been defined. Let's extend the above language to use operators and brackets. Let's construct ASTs calling functions with special names from operator usage. For example `11 + 2` is parsed into:

```
ast_application<
  ast_name<_S(".+.")>,
  ast_value<mpl::int_<11>>, ast_value<mpl::int_<2>>
>
```

The above code snippet turns operator `+` usage into `.+.` function calls. Half-constructed `meta_hs` blocks are types, thus one can create type aliases for them.

```
typedef meta_hs ::define<_S("f x = g x")>::type
                ::define<_S("g x = x")>::type my_lib;
```

The above code snippet defines two functions, `f` and `g` and creates a type alias for the resulting symbol table. As its name – `my_lib` – suggests, one can create template metaprogramming libraries this way. `my_lib` can be used the same way as `meta_hs` for defining further metafunctions:

```
typedef my_lib ::define<_S("h x = f x")>::type
               ::get<_S("h")>::type h;
```

The above code snippet defines a metafunction called `h` that uses `my_lib` and the `f` function provided by it. When operator calls, such as operator `+` are represented by special function calls, such as `.+.`, those function names need to be mapped to imported metafunctions implementing the operator evaluations. For example:

```
template <class A, class B>
struct lazy_plus :
  mpl::plus<typename A::type, typename B::type> {};

typedef meta_hs_base // some base class
  ::import2<_S(".+."), lazy_plus>::type
meta_hs;
```

The above code defines a function for `.+.` in `meta_hs`. It has to inherit from an empty symbol table, which is referred to as `meta_hs_base`. Due to the lack of lazy evaluation in Boost.MPL [25] a lazy version of the `mpl::plus` metafunction is needed. Using `import` can't make metafunctions automatically lazy, because it wouldn't work with metafunctions that are already lazy.

This section has presented an embedded DSL that is similar to Haskell and makes it possible to define template metafunctions using a more readable syntax.

V.3.7 Implementing factorial

The factorial example at the beginning of section V.3 can be implemented using the tools presented in this section. The steps of the implementation are the following:

- Start a new `meta_hs` block.
- Define the `fact` function.
- Export the `fact` function.

The following example implements it:

```
typedef meta_hs ::define<_S(

  "fact n ="
    " if n == 0"
      " then 1"
      " else n * fact (n-1)"

)>::type
  ::get<_S("fact")>::type fact;
```

This code snippet defines the function `fact` in an empty `meta_hs` block and exports it to construct the `fact` metafunction class, which can be used as any other manually constructed metafunction class. For example:

```
typedef
  fact::apply<mpl::int_<3>>::type
  factorial3;
```

This example uses `fact` to calculate the factorial of 3. The `apply` metafunction provided by Boost.MPL can also be used:

```
typedef
  mpl::apply<fact, mpl::int_<3>>::type
  factorial3;
```

This example calls `fact` using `mpl::apply` to calculate the factorial of 3.

V.4 Summary

This chapter has presented how to parse the content of string literals at compile-time using template metaprograms, how to use it for embedding domain specific languages into C++ without using external tools and how to provide a Haskell-like domain specific language for C++ template metaprogramming by compiling and executing them in the same compilation step.
Thesis 3: I have developed a method for implementing a parser generator library in C++ template metaprogramming. I have evaluated how it can be used for embedding domain specific languages into C++ and providing a more readable syntax for C++ template metaprogramming. None of these methods require external preprocessors. (chapter V)

Thesis 3.1: I have developed a method for turning string literals into character containers for C++ template metaprograms. Utilising this I have developed a method for implementing a parser generator library in C++. (section V.1)

Thesis 3.2: I have evaluated how parsers based on Thesis 3.1 can be used to embed domain specific languages into C++ without external preprocessors. (section V.2)

Thesis 3.3: I have developed a method based on Thesis 3.1 for providing a Haskell-like syntax for C++ template metaprograms without external preprocessors. (section V.3)

Table V.1: Related publications

	[53]	[54]	[62]	[65]	[66]	[74]
3.1	×		×	×	×	
3.2	×	×		×	×	×
3.3					×	

Chapter VI

Summary

In spite of the known similarities of template metaprogramming and the functional paradigm the current practice of template metaprogramming is still not based on this. This dissertation presented two approaches to develop template metaprograms following the functional paradigm. All of the techniques discussed are based on the C++ standard, they can be used with any standard compliant compiler. None of them requires external tools.

The first approach simulates basic functional language elements in template metaprogramming and builds higher-level abstractions on top of them. The new elements and techniques are built on top of the widely used elements of the Boost.MPL library, thus they can be easily adopted in programs already using that library.

Thesis 1: I have evaluated the connection between C++ template metaprogramming and functional programming languages. Based on the results I have developed methods for supporting template metaprogrammers using the functional paradigm explicitly. (chapter III)

Thesis 1.1: I have shown the importance of laziness in template metaprogramming and developed an automated adaption method to use non-lazy metafunctions in a lazy way. (section III.1)

Thesis 1.2: I developed a method for effective implementation of currying in C++ template metaprogramming. (section III.2)

Thesis 1.3: I have developed a method for representing Haskell-like algebraic data-types in C++ template metaprogramming. (section III.3)

Thesis 1.4: I have developed a method for representing Haskell typeclasses in C++ template metaprogramming. (section III.4)

Thesis 1.5: I have developed a method to handle template metaprogramming expressions as first class citizens, ie. they can be stored, passed as parameters or returned by functions. This method enables the implementation of let expressions and provides a more convenient way of implementing

lambda expressions than what Boost.MPL's lambda expression implementation, a widely used solution offers. (section III.5)

Thesis 1.6: I have implemented an alternative method for pattern matching in C++ template metaprogramming. This enables the implementation of case expressions. (section III.6)

Following the first approach, a common abstraction in functional languages, monads and a useful syntactic sugar, the do notation Haskell provides for monads are implemented in C++ template metaprogramming. A number of useful techniques can be built based on this. List comprehension can be provided for template metaprogramming based on the List monad. It makes list transformations easier to write, read, understand and maintain. Monads simplify error propagation in template metaprograms and this dissertation has shown how to simulate exception handling in template metaprogramming based on them.

Thesis 2: I have developed a method for implementing monads and a Haskell-like do syntax in C++ template metaprogramming and evaluated how a number of different monad variations available in Haskell can be implemented using this method. Based on this I have developed a method for simulating exception handling in C++ template metaprograms. (chapter IV)

Thesis 2.1: I have developed a method for implementing monads in C++ template metaprogramming. (section IV.1)

Thesis 2.2: I have evaluated how a number of monads available in Haskell can be implemented using the approach presented in Thesis 2.1. (section IV.2)

Thesis 2.3: I have developed a method for implementing a Haskell-like do syntax in template metaprogramming. (section IV.3)

Thesis 2.4: I have developed a method for simulating exception handling in C++ template metaprogramming based on monads. (section IV.4)

The other approach parses code snippets in string literals at compile-time and builds an interpreter for template metaprograms. Parsing a DSL snippet written in a string literal and its evaluation happens in the same compilation step, which makes it possible to provide a Haskell-like syntax for template metaprograms.

Thesis 3: I have developed a method for implementing a parser generator library in C++ template metaprogramming. I have evaluated how it can be used for embedding domain specific languages into C++ and providing a more readable syntax for C++ template metaprogramming. None of these methods require external preprocessors. (chapter V)

Thesis 3.1: I have developed a method for turning string literals into character containers for C++ template metaprograms. Utilising this I have developed a method for implementing a parser generator library in C++. (section V.1)

Thesis 3.2: I have evaluated how parsers based on Thesis 3.1 can be used to embed domain specific languages into C++ without external preprocessors. (section V.2)

Thesis 3.3: I have developed a method based on Thesis 3.1 for providing a Haskell-like syntax for C++ template metaprograms without external preprocessors. (section V.3)

All the techniques presented in this dissertation have been implemented in an open-source library collection [59]. People can download it and take advantage of the results presented in this dissertation. Table VI.1 generated using the Cloc [45] utility shows the number of lines of code of the libraries, their tests, the examples and the documentation.

Table VI.1: Lines of code in Mpllibs

Language	files	blank	comment	code
HTML	186	408	0	12383
C/C++ Header	232	2316	1159	10323
C++	182	2734	1446	8448
CMake	32	110	132	160
CSS	1	19	7	64
YAML	1	0	0	19

Appendix A

Summary

This dissertation introduces advanced techniques for C++ template metaprogramming supporting the developers and maintainers of applications and libraries implemented in C++. The connection between the functional programming paradigm and C++ template metaprogramming is well known. For C++ code executed at runtime there are libraries supporting functional programming but in template metaprogramming current approaches try to simulate imperative languages and libraries and most of them does not take advantage of the functional paradigm. This dissertation evaluates how the readability of template metaprograms can be improved by taking advantages of the functional nature of it. Two different approaches are discussed.

One of them extends the tools and techniques currently used in template metaprogramming with elements commonly used in functional programming languages. Among others it introduces algebraic data types, let expressions, pattern matching, currying and typeclasses in template metaprogramming. It also presents an implementation of monads which can be used to provide list comprehension and to simulate exception handling.

The other one is based on providing a Haskell-like DSL for template metaprogramming. Metaprograms written in that DSL are embedded into C++ in string literals and during the compilation of the host code they are transformed into template metaprograms, which are executed immediately. In order to achieve this, this dissertation discusses how to parse string literals with template metaprograms. It enables the smooth integration of DSLs into C++. This dissertation presents example applications of this technique.

All the techniques presented in this dissertation have been implemented in an open-source library collection. People can download it and take advantage of the results presented in this dissertation. The results and the library have been presented to the C++/Boost community. The lecture won the Best Presentation award on the C++Now conference, 2012, Aspen.

Appendix B

Összefoglalás

A dolgozat C++ template metaprogramozást segítő módszereket mutat be, melyek a C++ könyvtárak és alkalmazások fejlesztőit támogatják. A C++ template metaprogramozás kapcsolata a funkcionális paradigmával jól ismert. Futási időben végrehajtott C++ kód készítéséhez vannak könyvtárak, melyek a funkcionális programozást támogatják, viszont template metaprogramozásban a jelenleg használt módszerek az imperatív nyelveket és könyvtárakat szimulálják. Többségük nem használja ki a funkcionális paradigma által nyújtott lehetőségeket. A dolgozat megvizsgálja, hogy a template metaprogramok olvashatóságát milyen módon lehet a funkcionális paradigma mentén javítani. A dolgozat két módszert tárgyal.

Az egyik módszer a jelenleg használt eszközöket és módszereket bővíti funkcionális nyelvekben gyakori elemekkel. Többek között bevezeti az algebrai adattípusokat, let kifejezéseket, mintaillesztést, curry-zést és a typeclass fogalmát. Bemutatja továbbá, hogy hogyan lehet implementálni a monádokat, melyek segítségével megvalósítható a list comprehension, illetve szimulálható a kivételkezelés.

A másik módszer egy Haskell-szerű DSL-t valósít meg a template metaprogramozás számára. Az ezen a nyelven írt metaprogramokat karakterlánc literálokban lehet C++ kódba ágyazni. A C++ kód fordításakor ezek metaprogramokká lesznek alakítva és rögtön végrehajtásra kerülnek. Ennek megvalósításához a dolgozat bemutatja, miként lehet karakterlánc literálokat template metaprogramokkal feldolgozni. Ez lehetővé teszi a DSL-ek hatékony beágyazását C++-ba, melyre a dolgozat alkalmazási példákat mutat be.

A bemutatott módszerek egy nyílt forrású könyvtár gyűjteményben implementálásra kerültek, mely letölthető és a módszerek nyújtotta előnyök kihasználhatók. A dolgozat eredményei és a könyvtárak be lettek mutatva a C++/Boost közösségnek. Az előadás Best Presentation díjat nyert a C++Now konferencián 2012-ben Aspenben.

Bibliography

[1] Abrahams, D., and Gurtovoy, A. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004. ISBN: 0321227255.

[2] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN: 0321486811.

[3] Alexandrescu, A. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN: 0-201-70431-5.

[4] Alexandrescu, A. The loki library, 2006. http://loki-lib.sourceforge.net/.

[5] Andersson, L. Parsing with haskell, 2001. http://www.cs.lth.se/eda120/assignment4/parser.pdf.

[6] Armstrong, J. *Programming Erlang*, 1st ed. Pragmatic Bookshelf, 2007. ISBN: 9781934356005.

[7] Baron, P. *Erlang/OTP. Plattform für massiv-parallele und fehlertolerante Systeme*, 1st ed. Open Source Press, 2012. ISBN: 9783941841451.

[8] Bernardy, J.-P., Jansson, P., Zalewski, M., and Schupp, S. Generic programming with c++ concepts and haskell type classes: A comparison. *J. Funct. Program. 20*, 271–302.

[9] Booch, G. *Object-Oriented Analysis and Design with Applications (2nd Edition)*. Addison-Wesley Professional, September 1993. ISBN: 0805353402.

[10] Caro, M. Haskell to c++ template metaprogramming translator, 2010. http://code.google.com/p/phaskell/w/list.

[11] Csörnyei, Z. *Lambda-kalkulus*. Typotex Elektronikus Kiadó Kft., 2007. ISBN: 978-963-9664-46-3, 1787-3054.

[12] Czarnecki, K., and Eisenecker, U. W. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. ISBN: 0-201-30977-7.

[13] de Guzman, J., Kaiser, H., and Nuffer, D. Boost.spirit, 2003. http://www.boost.org/libs/spirit.

[14] Dybvig, R. K. *The Scheme Programming Language*. The MIT Press, 2009. ISBN: 978-0262512985.

[15] Érdi, G. Haskell to c++ template metaprogramming translator, 2010. http://gergo.erdi.hu/projects/metafun/.

[16] Fowler, M. *Domain-specific Languages*. Addison-Wesley, 2010. ISBN: 0321712943.

[17] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN: 0-201-63361-2.

[18] Gil, J. Y., and Lenz, K. Simple and safe sql queries with c++ templates. *Sci. Comput. Program. 75* (July 2010), 573–595.

[19] Golodetz, S. Functional programming using c++ templates (part 1). *Overload*, 81 (October 2007). http://www.accu.org/var/uploads/journals/overload81.pdf.

[20] Golodetz, S. Functional programming using c++ templates (part 2). *Overload*, 82 (December 2007). http://www.accu.org/var/uploads/journals/Overload82.pdf.

[21] Gregor, D., Järvi, J., Siek, J. G., Stroustrup, B., Reis, G. D., and Lumsdaine, A. Concepts: linguistic support for generic programming in c++. In *OOPSLA* (2006), P. L. Tarr and W. R. Cook, Eds., ACM, pp. 291–310.

[22] Gregor, D., Jeremy Siek, J. W., J arvi, J., Garcia, R., and Lumsdaine, A. Concepts for c++0x (revision 1). Technical Report N1849=05-0109, ISO/IEC JTC 1, Information Technology, Subcommittee 22, Programming Language C++, Aug. 2005.

[23] Gregor, D., and Siek, J. Implementing concepts. Technical Report N2617=08-0127, ISO/IEC JTC 1, Information Technology, Subcommittee 22, Programming Language C++, May 2008.

[24] Gregor, D., and Stroustrup, B. Wording for concepts (revision 1). Technical Report N2193=07-0053, ISO/IEC JTC 1, Information Technology, Subcommittee 22, Programming Language C++, 2007.

[25] Gurtovoy, A., and Abrahams, D. Boost.mpl, 2004. http://www.boost.org/libs/mpl.

[26] Haskellwiki. List comprehension. http://www.haskell.org/haskellwiki/List_comprehension.

[27] Horváth, Z., Plasmeijer, R., and Zsók, V., Eds. *Central European Functional Programming School - Third Summer School, CEFP 2009, Budapest, Hungary, May 21-23, 2009 and Komárno, Slovakia, May 25-30, 2009, Revised Selected Lectures* (2010), vol. 6299 of *Lecture Notes in Computer Science*, Springer.

[28] Hutton, G., and Meijer, E. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[29] Hutton, G., and Meijer, E. Monadic Parsing in Haskell. *Journal of Functional Programming 8*, 4 (July 1998), 437–444.

[30] ISO. *ISO/IEC 14882:2011 Information technology — Programming languages — C++*. International Organization for Standardization, Geneva, Switzerland, Feb. 2012.

[31] Iso14882. ISO/IEC 14882:1998: Programming languages – c++. Tech. rep., International Organization for Standardization, 1998.

[32] Josuttis, N. M. *The C++ standard library: a tutorial and reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[33] Karlsson, B. *Beyond the C++ Standard Library: An Introduction to Boost*. Addison Wesley Professional, Aug. 2005. ISBN: 0321133544.

[34] Karvonen, V., and Mensonides, P. Boost.preprocessor, 2001. http://www.boost.org/libs/preprocessor.

[35] Lutz, M. *Learning Python*, 3rd ed. O'Reilly Media, Inc., 2008. ISBN: 9780596513986.

[36] McNamara, B., and Smaragdakis, Y. Functional programming in c++ using the fc++ library. *SIGPLAN Notices 36*, 4 (2001), 25–30.

[37] Milewski, B. What does haskell have to do with c++?, 2009. http://bartoszmilewski.wordpress.com/2009/10/21/what-does-haskell-have-to-do-with-c/.

[38] Milewski, B. Monads for the curious progrmamer, 2011. http://bartoszmilewski.wordpress.com/2011/01/09/monads-for-the-curious-programmer-part-1/.

[39] Milewski, B. Monads in c++, 2011. http://bartoszmilewski.wordpress.com/2011/07/11/monads-in-c/.

[40] Muñoz, J. M. L. Monads in c++ template metaprogramming, 2008. http://bannalia.blogspot.com/2008/06/monads-in-c-template-metaprogramming.html.

[41] Musser, D. R., and Stepanov, A. A. Algorithm-oriented generic libraries. *Softw. Pract. Exper. 24* (July 1994), 623–642.

[42] Myers, N. *A new and useful template technique: "traits"*. SIGS Publications, Inc., New York, NY, USA, 1996, pp. 451–457.

[43] Niebler, E. Boost.proto, 2007. http://www.boost.org/libs/proto.

[44] Niebler, E. Boost.xpressive, 2007. http://www.boost.org/libs/xpressive.

[45] Northrop Grumman Corporation. Cloc - count lines of code, 2013. http://cloc.sourceforge.net/.

[46] Odersky, M., Spoon, L., and Venners, B. *Programming in Scala*. Artima Inc, Walnut Creek, CA, USA, 2010. ISBN: 978-0981531649.

[47] Okasaki, C. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, UK, 1999. ISBN: 0-521-66350-4.

[48] O'Sullivan, B., Goerzen, J., and Stewart, D. *Real World Haskell*, 1st ed. O'Reilly Media, Inc., 2008. ISBN: 0596514980, 9780596514983.

[49] Pepper, P., and Hofstedt, P. *Funktionale Programmierung*. Springer-Verlag, 2006. ISBN: 978-3-540-20959-1.

[50] Pickering, R. *Beginning F#*. Apress, 2009. ISBN: 978-1-4302-2389-4.

[51] Popa, D. How to build a monadic interpreter in one day. *Stud. Cercet. Stiint., Ser.Mat., Supplement Proceedings of CNMI 2007 17* (2007), 173–192.

[52] Porkoláb, Z. Functional programming with c++ template metaprograms. In Horváth et al. [27], pp. 306–353.

[53] Porkoláb, Z., and Sinkovics, Á. Domain-specific language integration with compile-time parser generator library. In *Generative Programming And Component Engineering, Proceedings of the Ninth International Conference on Generative Programming and Component Engineering, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010* (2010), E. Visser and J. Järvi, Eds., ACM, pp. 137–146.

[54] Porkoláb, Z., Sinkovics, Á., and Siroki, I. Dsl in c++ template metaprogram, tutorial, 2013. http://dsl2013.math.ubbcluj.ro/files/Lecture/PorkolabEtAl_TemplateMetaprogramming.pdf.

[55] Ramsey, N. Eliminating spurious error messages using exceptions, polymorphism, and higher-order functions. *Computer Journal 42* (1999).

[56] Sheard, T., Benaissa, Z.-e.-a., and Pasalic, E. Dsl implementation using staging and monads. *SIGPLAN Not. 35* (December 1999), 81–94.

[57] Sinkovics, Á. Functional extensions to the boost metaprogram library. *Electr. Notes Theor. Comput. Sci. 264*, 5 (2010), 85–101.

[58] Sinkovics, Á. Functional extensions to the boost metaprogram library. In *WGT'10* (2010), Z. Porkoláb and N. Pataki, Eds., vol. II of *WGT Proceedings*, Zolix, pp. 56–66.

[59] Sinkovics, Á. The source code of mpllibs, 2010. http://github.com/sabel83/mpllibs.

[60] Sinkovics, Á. Nested lamda expressions with let expressions in c++ template metaprorgams. In *WGT'11* (2011), Z. Porkoláb and N. Pataki, Eds., vol. III of *WGT Proceedings*, Zolix, pp. 63–76.

[61] Sinkovics, Á. Boosting mpl with haskell elements, 2013. http://www.youtube.com/watch?v=aIj034VCUD8.

[62] Sinkovics, Á., and Abrahams, D. Using strings in c++ template metaprograms, 2012. http://cpp-next.com/archive/2012/10/using-strings-in-c-template-metaprograms/.

[63] Sinkovics, Á., and Porkoláb, Z. Expressing c++ template metaprograms as lamda expressions. In Horváth et al. [27], pp. 97–111.

[64] Sinkovics, Á., and Porkoláb, Z. Implementing monads for c++ template metaprograms. Technical Report TR-01/2011, Eötvös Loránd University, Faculty of Informatics, Dept. of Programming Languages and Compilers, Sept. 2011.

[65] Sinkovics, Á., and Porkoláb, Z. Domain-specific language integration with c++ template metaprogramming. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments* (2012), 32. ISBN: 1466620927.

[66] Sinkovics, Á., and Porkoláb, Z. Metaparse - compile-time parsing with c++ template metaprogramming, 2012. http://cppnow.org/files/2012/04/Sinkovics.Porkol%C3%A1b.pdf.

[67] Sinkovics, Á., and Porkoláb, Z. Implementing monads for c++ template metaprograms. *Science of Computer Programming 78*, 0 (2013), 1600 – 1621.

[68] Sipos, Á., Porkoláb, Z., and Zsók, V. Meta¡fun¿ - towards a functional-style interface for c++ template metaprograms. *Studia Universitatis Babes-Bolyai Informatica LIII*, 2008/2 (2008), 55–66.

[69] Spivey, M. A functional theory of exceptions. *Sci. Comput. Program. 14* (May 1990), 25–42.

[70] Steele, Jr., G. L. *Common LISP: the language (2nd ed.)*. Digital Press, Newton, MA, USA, 1990. ISBN: 1-55558-041-6.

[71] Stroustrup, B. Simplifying the use of concepts. Technical Report N2906=09-0096, ISO/IEC JTC 1, Information Technology, Subcommittee 22, Programming Language C++, 2009.

[72] Stroustrup, B. *The C++ Programming Language*, 4th ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2013. ISBN: 0321563840.

[73] Stroustrup, B., and Reis, G. D. Concepts - design choices for template argument checking. Technical Report N1522=03-0105, ISO/IEC JTC 1, Information Technology, Subcommittee 22, Programming Language C++, Oct. 2003.

[74] Szűgyi, Z., Sinkovics, Á., Pataki, N., and Porkoláb, Z. C++ metastring library and its applications. In *GTTSE* (2009), J. M. Fernandes, R. Lämmel, J. Visser, and J. Saraiva, Eds., vol. 6491 of *Lecture Notes in Computer Science*, Springer, pp. 461–480.

[75] Unruh, E. Prime number computation, 1994.
ANSI X3J16-94-0075/ISO WG21-462.

[76] Vandevoorde, D., and Josuttis, N. M. *C++ Templates: The Complete Guide*, 1 ed. Addison-Wesley Professional, Nov. 2002. ISBN: 9780201734843.

[77] Veldhuizen, T. Expression templates. *C++ Report 7* (1995), 26–31.

[78] Veldhuizen, T. *Using C++ template metaprograms*. SIGS Publications, Inc., New York, NY, USA, 1996, pp. 459–473.

[79] Veldhuizen, T. L. C++ templates are turing complete. Tech. rep., 2003.

[80] Veldhuizen, T. L., and Gannon, D. Active libraries: Rethinking the roles of compilers and libraries. In *In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing OO'98* (1998), SIAM Press.

[81] Wikipedia. Evaluation strategy, 2013.
http://en.wikipedia.org/wiki/Evaluation_strategy.

[82] Wikipedia. Thunk (functional programming), 2013. http://en.wikipedia.org/wiki/Thunk_%28functional_programming%29.

[83] Zólyomi, I., Porkoláb, Z., and Kozsik, T. An extension to the subtype relationship in c++ implemented with template metaprogramming. In *Generative Programming and Component Engineering*, F. Pfenning and Y. Smaragdakis, Eds., vol. 2830 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003, pp. 209–227.